Addison-Wesley

STEVE WILLIAMS

68030 ASSEMBLY LANGUAGE REFERENCE

INCLUDES THE 68020

68030
Assembly Language
Reference

68030 Assembly Language Reference

Including the 68020

STEVE WILLIAMS

Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam •
Bonn Sydney • Singapore • Tokyo • Madrid • San Juan

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters.

Library of Congress Cataloging-in-Publication Data

Williams, Steve.
68030 assembly language reference/Steve Williams
p. cm.
Includes index.
ISBN 0-201-08876-2
1. Motorola 68030 (Microprocessor)—Programming. 2. Assembler language (Computer program language) I. Title.
QA76.8.M6898W55 1989 004.165—dc20 89-33101

Copyright © 1989 by Steve Williams

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Sponsoring Editor: Carole McClendon
Cover design by Doliber Skeffington
Text design by Total Concept Associates
Set in 10-point Bookman by Desktop Publishing, Consultation & Services

ABCDEFGHIJ-HA-89
First printing, August 1989

Dedicated with love to Annarine Hamilton.

Acknowledgments

> *"If I have seen farther than other men,*
> *it is because I stood on the shoulders of giants."*
> *— Sir Isaac Newton*

I would like to thank the following people, without whom this book would not exist. I would not have even started were it not for the gentle insistence of my good friend, Carole McClendon of Addison-Wesley.

Thanks to Bruce Montague for a thorough, painstaking technical review. In addition to locating a number of technical flaws, Bruce also made a number of important suggestions concerning content and style. This book is far better than it might otherwise have been as a result.

Thanks to Carole McClendon, Joanne Clapp, Rachel Guichard, Diane Freed, Perry McIntosh, and Beth Burleigh for their able administration of the writing and production processes.

Thanks to Leslie Bartlett of Desktop Publishing, Consultation & Services for his incredible effort in making 70% reductions in the illustrations, despite uncooperative software.

Thanks to my brother, Jim Williams, for the use of his Macintosh Plus in Florida during Christmas vacation.

This book was written using Apple Macintosh II, Macintosh Plus, and LaserWriter hardware, with Microsoft Word and MacDraw software. The vendors of these products are to be commended for their contributions to the publishing industry.

Thanks to all.

CONTENTS

CHAPTER 1. **Introduction** 1

Background 1
Organization 1
Notational Conventions 1
    Assembly Language Syntax 2
    Addressing Mode Differences 2
    Register Restrictions 4
    Hexadecimal Conventions 4
Summary 5

CHAPTER 2. **Architecture** 7

Introduction 7
68000 Family Evolution 7
    68000 8
    68010 8
    68020 9
    68030 10
Data Types 10
Register Set 12
    Data Registers 13
    Address Registers 13
    PC 13
    CCR 14
    SR 14
    Processor Modes 15
    MSP and ISP 16
    VBR 16
    CAAR and CACR 17
    SFC and DFC 17
Addressing Modes 18
    Data Register Direct 19
    Address Register Direct 19
    Address Register Indirect 19
    Address Register Indirect with Postincrement 20
    Address Register Indirect With Predecrement 20

Address Register Indirect With Displacement 21
Address Register Indirect With Index (8-bit Displacement).22
Address Register Indirect With Index (Base Displacement).23
Memory Indirect Post-Indexed 25
Memory Indirect Pre-Indexed..27
Program Counter Indirect with Displacement 28
Program Counter Indirect with Index (8-bit displacement) .28
Program Counter Indirect With Index (Base Displacement)30
Program Counter Memory Indirect Post-Indexed................32
Program Counter Memory Indirect Pre-Indexed 34
Absolute Short Address 35
Absolute Long Addressing 35
Immediate Data...36
Memory Cache Operation..36
68020 Cache 37
68030 Cache 39
Exception Processing 44
Exception Stack Frames..45
Faults...46
Interrupts...58
Coprocessor Operations 61
Coprocessor Instruction Format..62
Coprocessor-Generated Exceptions 63

CHAPTER 3. Instruction Set

Instruction Set...65
Introduction..65
Format...65
Header 65
Description...66
Addressing Modes 67
Instruction Format...68
Example 68
ABCD Instruction 70
ADD Instruction..72
ADDA Instruction 74
ADDI Instruction...76
ADDQ Instruction 78
ADDX Instruction 80
AND Instruction..82
ANDI Instruction...84
ANDI to CCR Instruction...86
ANDI to SR Instruction (Privileged)...88

ASL Instruction....90
ASR Instruction....93
Bcc Instruction....96
BCHG Instruction....99
BCLR Instruction....101
BFCHG Instruction....103
BFCLR Instruction....106
BFEXTS Instruction....109
BFEXTU Instruction....112
BFFFO Instruction....115
BFINS Instruction....118
BFSET Instruction....121
BFTST Instruction....124
BKPT Instruction....127
BSET Instruction....129
BSR Instruction....131
BTST Instruction....133
CALLM Instruction....135
CAS Instruction....141
CAS2 Instruction....144
CHK Instruction....146
CHK2 Instruction....148
CLR Instruction....151
CMP Instruction....153
CMPA Instruction....156
CMPI Instruction....159
CMPM Instruction....162
CMP2 Instruction....164
cpBcc Instruction....167
cpDBcc Instruction....168
cpGEN Instruction....169
cpRESTORE Instruction (Privileged)....170
cpSAVE Instruction (Privileged)....172
cpScc Instruction....174
cpTRAPcc Instruction....176
DBcc Instruction....177
DIVS Instruction....180
DIVSL/DIVS.L Instructions....182
DIVU Instruction....185
DIVUL/DIVU.L Instructions....188
EOR Instruction....191
EORI Instruction....193
EORI to CCR Instruction....196

EORI to SR Instruction (Privileged) 198
EXG Instruction 200
EXT Instruction 202
EXTB Instruction 204
Illegal Instruction 206
JMP Instruction 207
JSR Instruction 209
LEA Instruction 211
LINK Instruction 213
LSL Instruction 215
LSR Instruction 218
MOVE Instruction 221
MOVEA Instruction 223
MOVE From CCR Instruction 225
MOVE To CCR Instruction 227
MOVE From SR Instruction (Privileged) 229
MOVE To SR Instruction (Privileged) 231
MOVE USP Instruction (Privileged) 233
MOVEC Instruction (Privileged) 235
MOVEM Instruction 237
MOVEP Instruction 241
MOVEQ Instruction 244
MOVES Instruction (Privileged) 246
MULS Instruction 249
MULS.L Instructions 251
MULU Instruction 254
MULU.L Instructions 256
NBCD Instruction 259
NEG Instruction 262
NEGX Instruction 264
NOP Instruction 266
NOT Instruction 268
OR Instruction 270
ORI Instruction 272
ORI to CCR Instruction 274
ORI to SR Instruction (Privileged) 276
PACK Instruction 278
PEA Instruction 280
RESET Instruction (Privileged) 282
ROL Instruction 283
ROR Instruction 286
ROXL Instruction 289
ROXR Instruction 292

RTD Instruction....................295
RTE Instruction (Privileged)....................297
RTM Instruction....................300
RTR Instruction....................302
RTS Instruction....................304
SBCD Instruction....................305
Scc Instruction....................307
STOP Instruction (Privileged)....................310
SUB Instruction....................312
SUBA Instruction....................314
SUBI Instruction....................316
SUBQ Instruction....................318
SUBX Instruction....................321
SWAP Instruction....................324
TAS Instruction....................325
TRAP Instruction....................328
TRAPcc Instruction....................330
TRAPV Instruction....................333
TST Instruction....................335
UNLK Instruction....................337
UNPK Instruction....................339

CHAPTER 4. Floating Point Instructions....................343

Background....................343
Architecture....................343
    Data Formats....................344
    Register Set....................350
Floating Point Conditional Operations....................356
Floating Point State Frames....................360
Floating Point Exceptions....................364
Concurrency Considerations....................373
System Software Considerations....................378
Instruction Description Format....................380
    Header....................380
    Description....................381
    Numerical Results....................382
    Addressing Modes....................382
    Instruction Format....................383
    Example....................384
FABS Instruction....................386
FACOS Instruction....................389
FADD Instruction....................392

FASIN Instruction 395
FATAN Instruction 398
FATANH Instruction....................401
FBcc Instruction 404
FCMP Instruction....................407
FCOS Instruction....................410
FCOSH Instruction 413
FDBcc Instruction....................416
FDIV Instruction 419
FETOX Instruction....................422
FETOXM1 Instruction....................425
FGETEXP Instruction....................428
FGETMAN Instruction....................431
FINT Instruction 434
FINTRZ Instruction 437
FLOG10 Instruction....................440
FLOG2 Instruction....................443
FLOGN Instruction....................446
FLOGNP1 Instruction....................449
FMOD Instruction....................452
FMOVE Instruction (Control Reg) 455
FMOVE Instruction (Data Reg) 458
FMOVECR Instruction 464
FMOVEM Instruction (Control Reg) 467
FMOVEM Instruction (Data Reg) 470
FMUL Instruction....................474
FNEG Instruction....................477
FNOP Instruction....................480
FREM Instruction 481
FRESTORE Instruction (Privileged)....................485
FSAVE Instruction (Privileged)....................488
FSCALE Instruction 491
FScc Instruction....................494
FSGLDIV Instruction....................497
FSGLMUL Instruction 500
FSIN Instruction 503
FSINCOS Instruction 506
FSINH Instruction....................510
FSQRT Instruction....................513
FSUB Instruction....................516
FTAN Instruction 519
FTANH Instruction....................522
FTENTOX Instruction....................525

FTRAPcc Instruction 528
FTST Instruction 531
FTWOTOX Instruction 534

CHAPTER 5. Memory Management Support 537

Background 537
What is a Memory Management Unit? 537
Memory Management Techniques 538
    Base/Bounds Memory Management 539
    Address Translation Cache 541
Motorola Memory Management Unit Architecture 542
    68451 543
    68851 543
    68030 544
    Translation Table Overview 544
    Register Set 546
    Translation Table Descriptors 557
    MMU Conditional Operations 565
    MMU State Frames 566
    MMU-Generated Exceptions 568
Instruction Description Format 570
    Header 571
    Description 571
    Addressing Modes 572
    Instruction Format 572
PBcc Instruction (Privileged) 575
PDBcc Instruction (Privileged) 578
PFLUSH Instruction (Privileged) 581
PFLUSHR Instruction (Privileged) 585
PLOAD Instruction (Privileged) 588
PMOVE Instruction (Privileged) 591
PRESTORE Instruction (Privileged) 595
PSAVE Instruction (Privileged) 597
PScc Instruction (Privileged) 600
PTEST Instruction (Privileged) 603
PTRAPcc Instruction (Privileged) 607
PVALID Instruction (Privileged) 610

CHAPTER 6. Program Examples 613

Introduction 613
Interfacing to a High-Level Language 614
    Apple MPW C 614

UNIX pcc Compiler 621
Macintosh Pascal 623
Performance Improvements 626
Programming Examples 627
C String Functions 627
Loop Unrolling Examples 632
qsort 640
bsearch 648
Graphics Examples 652
Floating Point Examples 661
Solution of Simultaneous Linear Equations 665
Miscellaneous 673
Instruction Tracer 676
Conclusion 683

APPENDIX A. **UNIX Assembly Language 685**

Introduction 685
Overview 685
Program Listings 687
strcpy 687
strcmp 688
_memcpy 689
_memclr 691
_memcmp 692
qsort 694
bsearch Routine 699
line Routine 700
circle Routine 701
bisect Routine 703
gauss Routine 704
is020 Routine 708
config Routine 708
tracer Routine 709

APPENDIX B. **Glossary 715**

Introduction 715
Definitions 715

Index 743

CHAPTER

1

Introduction

"He who has begun is half done". — Horace

Background

Programming in any language is a detail-intensive activity, and assembly language programming requires more attention to minute details than almost any other software discipline. The amount of information required to author even a small assembly language program overwhelms the memory capacity of the average human being.

A mature CPU architecture compounds this difficulty. A chip like the 68020 or 68030 is constrained by the requirement for compatibility with older versions of the family. Thus, the chip designers were not free to change any of the existing instruction set to mesh with the newer instructions added for these chips. As a result, the results are sometimes idiosyncratic.

Another problem with assembly language programming is that information is often required in random order. The user frequently needs to locate some minor facet of an obscure instruction and cannot progress until finding it in a reference. To alleviate this situation, we have organized the information so that you can easily locate necessary details. The book covers the CPU instruction set, as well as the two most common add-on chips, the 68881 floating point processor and the 68851 MMU.

Organization

This book is organized as follows:

Chapter 2 contains an overview of the 68000 family architecture, including the techniques for interfacing the CPU with the add-on chips, which are called *coprocessors*. This chapter also covers material usually not included in an architectural discussion such as the interrupt structure and processor traps.

Chapter 3 contains 68000 family instruction set descriptions that apply to all 68000 family members. Special note is made of any differences between family members. This text will enable you to write code that runs on all 68000 family chips, as well as to write code that takes advantage of features present on the more advanced models.

Chapter 4 covers the 68881 and 68882 floating point coprocessors. These chips are rapidly becoming standard equipment on desktop machines such as the Macintosh II. Software developers can therefore depend on the presence of the additional features supplied by these chips. Use of the coprocessor permits higher performance in some applications.

Chapter 5 describes the 68851 Memory Management Unit (MMU). With the advent of the 68030 CPU (which has a built-in MMU), memory protection will become far more commonplace. Many more systems will provide *virtual memory*, which allows running programs that are larger than the machine's memory.

Finally, Chapter 6 is a series of sample programs that illustrate subtle tricks to improve performance on 68000 family machines.

Notational Conventions

Several machines such as the Apple Macintosh line, the ST series from Atari, the Commodore Amiga series, and several models of Sun workstations use 68000 family CPU chips. These machines use a wide variety of development tools and operating systems. Thus, we have attempted to write example programs that will run on any environment.

There are, however, differences among these machines that cannot be overlooked. These differences are in the areas of assembly language syntax, addressing modes allowed, use of privileged instructions, and restrictions on the use of certain registers. As a result, the example programs may not always run on particular machines.

Assembly Language Syntax

As is often the case with mature CPU products, 68000 computers support a variety of different assembly language development environments. Unfortunately, each tool has its own dialect of the assembly source language. These tools fall into two major camps: those that follow the Motorola standard, and those that follow the UNIX standard. There are, of course, also differences within each group.

Motorola prescribed a standard assembly language when the 68000 chip was introduced. Many development tool authors selected this standard since it is used by the Motorola CPU documentation. You will find the Motorola syntax on the Apple Macintosh series, the Atari ST series, and the Commodore Amiga series. There are several differences between these machines, and we have tried to write the examples in this book so that they will run on all of them. Since the programs were developed and tested on the Apple Macintosh Programmer's Workbench (MPW), that syntax is chosen where these systems clash.

Most UNIX systems use “MIT syntax,” which was originally used by an early 68000 assembler done at MIT. The assembler used for Sun systems, the Motorola-supplied UNIX system, and the public domain GNU system all fall into this category.

The Apple A/UX system, while in most respects a traditional UNIX system, offers an interesting approach. The A/UX assembler supports both the Motorola and UNIX syntax, although the A/UX assembly language syntax differs from standard Motorola syntax in many important respects.

Addressing Mode Differences

Perhaps a more severe incompatibility stems from arbitrary restrictions placed on the programmer by certain systems, notably the Macintosh. The 68000 family has three basic techniques for addressing memory: by absolute address, relative to a register, and relative to the program counter. Motorola designed the 68000 so that the absolute addressing modes are almost mandatory. The register-relative forms require you to tie up a register, and were originally good for areas of less than 32,768 bytes. The PC-relative forms may not be used to store data into memory, and also were originally good for a range of plus or minus 32,768.

The absolute modes place absolute memory addresses inside the instructions. Thus, if you load a program in a place other than where it was linked, you must adjust references to items inside the program. (References to addresses outside the program, like low memory or hardware addresses, do not require adjustment.) The Macintosh software that loads a program into memory does not perform this operation, and absolute addresses of data inside the program are not allowed. The net effect is that Macintosh programs cannot have more than 32,768 bytes in any single program segment. This needlessly constrains all programming languages for the Macintosh.

Programming examples in this book attempt to steer clear of this conflict as well. Where this is not possible, the differences are noted so that the code will be usable on as many systems as possible.

Register Restrictions

The Macintosh programming model imposes yet another constraint in the area of register usage. Register A5 on the Macintosh points to the application's global data area, and most of the Macintosh system software requires that this register retain this value. You can change A5, but you must restore it before calling any of the Macintosh system software routines.

This technique is an unfortunate choice because of the burden it places on compiler writers and application developers. Compilers and applications essentially lose one of the 68000 registers. Adapting Macintosh code to another system, or adapting another system's code to the Macintosh is difficult. The most serious ramification of this design is that applications become limited to 32K of static (initialized) data. We will avoid this restriction where possible as well.

Hexadecimal Conventions

Hexadecimal notation is widely used throughout this book. Hex numbers will conform to the Motorola convention in which a dollar sign character ($) precedes each hex number (for instance, the hex number FF (255 decimal) appears as $FF). Diagrams of memory and register layouts are shaded to show hexadecimal digit boundaries where appropriate. The 68000 family instruction fields are particularly ill-behaved in this respect, and instruction fields almost never align to hexadecimal digit boundaries.

Summary

This book is intended for reference purposes during the programming process. The emphasis is on locating important details quickly. Differences between members of the 68000 family are also noted where appropriate, making it possible to write programs that run on all 680x0 processors.

The development environment is also a consideration. Although we have tried to make this book applicable to as many environments as possible, this is not always possible. Where a conflict exists, we will be compatible with the Macintosh MPW programming environment.

CHAPTER

2

Architecture

> *"Total grandeur of a total edifice,*
> *Chosen by an inquisitor of structures."*
> — *Wallace Stevens*

Introduction

This chapter covers the underlying architecture of the entire 68000 family, including the 68000 family evolution, register set, addressing modes, and other concepts basic to these CPU chips. Advanced topics, such as exceptions, cache control, and special purpose registers are also covered.

68000 Family Evolution

Let's start with a history lesson. Figure 2-1 illustrates the evolution of these CPU products, from the introduction of the 68000 in 1979 to the introduction of the 68030 in 1987.

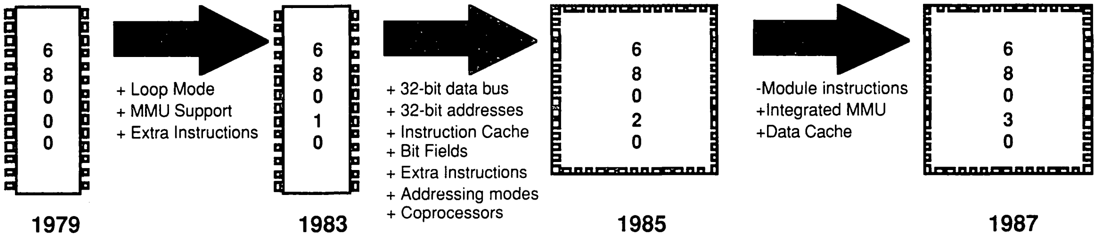

68000
1979
+ Loop Mode
+ MMU Support
+ Extra Instructions
68010
1983
+ 32-bit data bus
+ 32-bit addresses
+ Instruction Cache
+ Bit Fields
+ Extra Instructions
+ Addressing modes
+ Coprocessors
68020
1985
-Module instructions
+Integrated MMU
+Data Cache
68030
1987

Figure 2-1. 68000 Family Evolution

68000

The original member of this family, the 68000 was introduced in 1979. This chip has a 24-bit address bus, 16 32-bit registers, and a linear address space. The ability to linearly address 16Mb of memory (24 bits) contrasted sharply with the segmented address spaces of the chief competitors at the time, the Intel 8086 and the Zilog Z8000, which divide memory into 64K segments. The 68000's non-segmented address space greatly simplifies writing software for this chip. The architects of the 68000 demonstrated vision by allowing the 24-bit addresses in the 68000 to expand easily to 32 bits in the 68020. Nearly all contemporary microprocessor designs now offer at least a 32-bit address space.

68010

The 68010 added some performance enhancements, operating system support features, and new instructions. The biggest performance improvement is a *loop mode* feature. On the 68000, copying an area of memory typically requires a two-instruction loop. Four memory references are required to copy a memory item: one data fetch, one data store, and two instruction fetches. On the 68000, half of the memory bandwidth is spent repeatedly fetching those same two instructions. Loop mode eliminates these instruction fetches for certain two-instruction loops reducing the number of memory references to two.

MMU support is a major enhancement on the 68010. The 68000 lacks the ability to restart an instruction following a memory fault, but the 68010 permits the instruction to be continued after the condition that caused the fault has been corrected. This hardware feature is essential to support a software feature called *virtual memory*. Virtual memory permits running software that requires more memory than exists in the machine. There are also extra registers and instructions that allow movement back and forth between different memory spaces mapped by an MMU. This MMU support is vital for large operating systems like UNIX.

The 68010 also includes an additional subroutine return instruction. This new return instruction deallocates subroutine parameters and performs the return. This technique reduces code size, particularly for languages like PASCAL that do not require variable-length argument lists.

68020

The 68020 represents a major enhancement to the 68000 and 68010. The two older CPU chips had 16-bit data paths and 24-bit addresses. The 68020 expands both addresses and data to 32 bits. The address expansion allows the 68020 to address 4Gb (4,294,967,296 bytes), in contrast to the 16Mb (16,777,216 bytes) capability of the 68000 and 68010. The 32-bit data bus means that the processor can access a 32-bit quantity in a single memory access as opposed to the two accesses required by the older CPU chips.

Another major feature on the 68020 is the *instruction cache.* The 68020 retains the most recently used 256 bytes of the instruction stream in an on-board memory called the *cache.* Retaining these instructions effectively expands the 68010 loop mode to encompass a much larger set of program loops. Cache memory is extremely important when doing graphics and other computationally intensive software. Much of the performance of the Macintosh II and Sun 3 computer graphics is due to the 68020 cache.

The 68020 features a new data type called a *bit field,* which consists of a series of bits at an arbitrary bit boundary in memory. Hardware manipulation of bit fields is also extremely important to computer graphics.

Extra instructions and addressing modes add significant power to the 68020. The 16-bit limitations present in many of the 68000 instructions have been raised to 32 bits for the 68020. There are new instructions for bit field manipulation, module call/return, multiple processor support, and BCD conversion. Indexing and scaling have been added to some addressing modes, thus permitting faster access to array elements. (These operations allow use of a register for a subscript and adjust the subscript to the required data size.) The net effect of the instruction and addressing mode enhancement allows programs to be both smaller and faster by requiring fewer instructions.

Coprocessors are a new feature on the 68020. A coprocessor is a hardware device that adds new instructions to the 68020. The 68881 and 68882 floating point coprocessors (described in Chapter 4) are the best examples of this technique. A machine with a 68881/2 installed executes floating point instructions in hardware; a machine without this chip must use software techniques to perform floating point operations. There is usually a 10:1 performance improvement using a 68881/2 to do floating point. This capability is important to scientific and 3-D graphics applications.

68030

The 68030 is an integration of the 68020 and an associated chip, the 68851 Paged Memory Management Unit (PMMU). The 68030 does not contain all the features of the 68020/68851 combination. The module call/return instructions (CALLM/RETM) and several PMMU features have been omitted.

Data Types

The 68020 supports the following data types:

- 8-bit *bytes*
- 16-bit *words*
- 32-bit *longwords*
- Single *bits*
- Variable length *bit fields*
- Binary-coded decimal

A 68020 machine equipped with a 68881/2 floating point unit also supports several other data types. These are described in Chapter 4.

The 8-, 16-, and 32-bit quantities are the most common. Figure 2-2 illustrates the relationship between these data types.

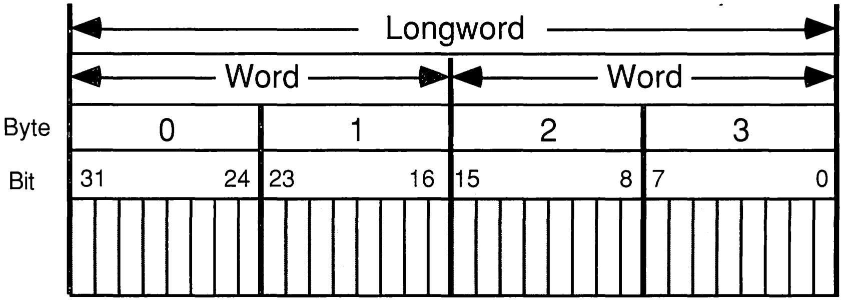


Figure 2-2. 68020 Integer Data Types

A *longword* is a 32-bit binary number. On the 68000 and 68010, words and longwords must begin on even addresses. Longwords on the 68020 and 68030 may begin at any byte address, but these newer machines give better performance if longwords begin at addresses evenly divisible by 4. A *word* is a 16-bit quantity, or half a longword. On the 68020 and 68030, words may also begin at any byte boundary, but performance is better when words begin on even addresses. A *byte* is an 8-bit quantity.

On 68000 machines, the lowest address in a word or longword contains the most significant byte of the integer; this is called *big endian* order. IBM mainframe computers (360, 370, 4300, and so on) also use this byte ordering. Most popular computers use the opposite ordering, in which the least significant byte is found at the lowest address in these machines; this is called *little endian* order. The Intel 80x86 (used in the IBM PC family) and the DEC VAX-11 are examples of little endian machines.

A *bit* data type is a single bit within a byte, specified by the combination of a byte address and a bit number within that byte. A bit may have only two possible values, 0 or 1.

A *bit field* is a series of bits, beginning with an arbitrary bit in a byte and extending from 1 to 32 bits. The field is identified by an offset and a width. The offset is the number of bits to the left of the most significant bit in the field, and the width is the number of bits in the field. Bit fields are extremely useful in graphics software. Figure 2-3 contains an illustration of a bit field.

This field has an offset of 6 and a size of 11. Note that the offset is the number of bits from the left; most other bit numberings are from the right. The 68000 and 68010 do not support bit fields.

The *Binary-Coded Decimal* (BCD) data type is widely used in accounting and financial applications. BCD numbers are decimal integers stored two digits per byte, as shown in Figure 2-4.

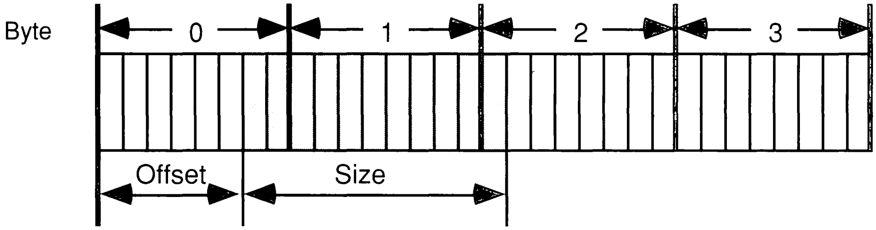


Figure 2-3. Bit Field Layout

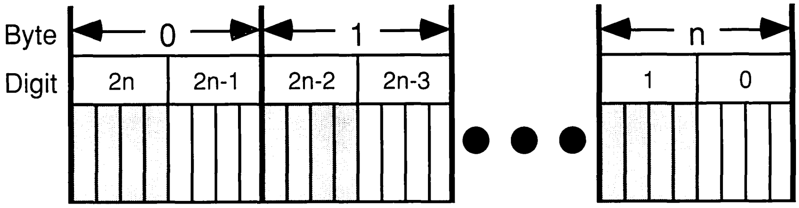

Byte
0
1
n
Digit
2n
2n-1
2n-2
2n-3
1
0

Figure 2-4. Binary-Coded Decimal (BCD)

Each byte is divided into two 4-bit groups. These 4-bit fields hold values from 0–9. The remaining six values (10–15 decimal or A–F hex) are unused. BCD numbers are also stored in big endian order.

Register Set

Figure 2-5 illustrates the registers available to 68020 software. Normal (user) software has access to eight 32-bit data registers, eight 32-bit address registers, a program counter (PC), and a condition code register (CCR). Supervisor (privileged) software has access to several additional registers.

Register names in boldface exist only on the 68010 and later processors. Registers whose names are underlined exist only on

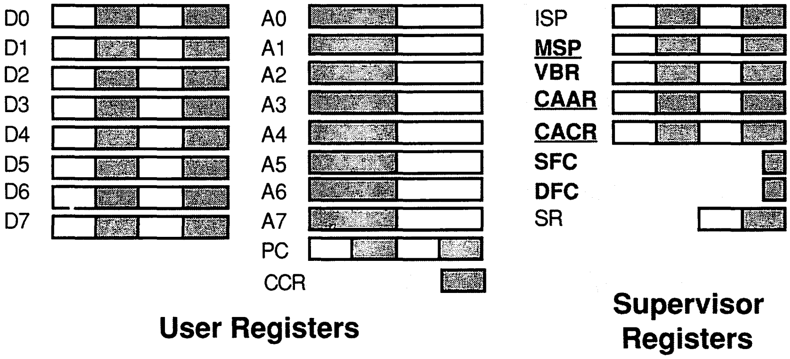

D0
D1
D2
D3
D4
D5
D6
D7
A0
A1
A2
A3
A4
A5
A6
A7
PC
CCR
User Registers
ISP
MSP
VBR
CAAR
CACR
SFC
DFC
SR
Supervisor Registers

Figure 2-5. 68020 Registers

the 68020 and later processors. Most registers are composed of individual bytes, as indicated by the shaded portions above. The address registers and the SFC/DFC pair are the exceptions, as explained below.

Data Registers

There are eight 32-bit data registers named D0–D7. You can access or change the low 8 bits, low 16 bits, or all 32 bits of a data register. Changing the low byte or low word of the data register has no effect on the remaining portion. Several instructions require one or more operands to reside in a data register.

Address Registers

There are eight 32-bit address registers, named A0–A7. You can access either the low-order 16 bits of an address register or all 32 bits. Changing an address register always affects all 32 bits. Changing the low word of the register replicates bit 15 in bits 16–31. This is known as *sign extension*, which converts a two's complement 16-bit quantity to an equivalent 32-bit quantity.

Register A7 serves a special purpose as the 68020 *stack pointer* register. You can refer to this register as either A7 or SP in your assembly language programs. The stack is a storage area for data that is needed only on a temporary basis. Items are added to the stack by decrementing the stack pointer and storing the item; items are removed by incrementing the stack pointer. Thus, A7 always points to the last item placed on the stack. This storage technique is called “Last In First Out,” or LIFO, because the first item removed from a stack is the last item added. The address in A7 must always be even.

PC

The *program counter* register, or PC, controls program execution. The PC contains the address of the next instruction to be executed. This address advances as each instruction executes. Special instructions can change the value of the program counter depending on the result of previous computations. Instructions must always begin on even addresses.

CCR

The results of previous computations are stored in a series of five bits called *condition codes.* These bits are stored in a 16-bit register called the *Condition Code Register,* or *CCR.* This register is shown in Figure 2-6.

Only the low-order 5 bits in this register are defined. All others read back as zero and are ignored on write. The X (eXtend) bit is used as a carry in multiple precision add, subtract, compare, negate, and shift operations. The N (Negative) bit is copied from the high-order bit of the result of an operation. This allows testing for values less than zero (using two's complement). The Z (Zero) bit is set if all bits of the result of an operation are zero. The V (oVerflow) bit is set when the result of an operation is too large to fit in the destination operand. The C (Carry) bit is set whenever there is a carry out of the most significant bit of an operation. As explained in Chapter 3, these bits are set differently by different instructions.

The CCR register can be modified but not read on the 68000 chip. On all other members of the family, the CCR can be both read and modified.

SR

The CCR is actually the low-order byte of a 16-bit register called the *Status Register,* or *SR.* Programs on the 68000 must use the status register (SR) to read the condition code bits. On chips other than the 68000, reading the status register is limited to privileged software. This situation requires special software on the later chips to simulate user mode access to the status register on the 68000.

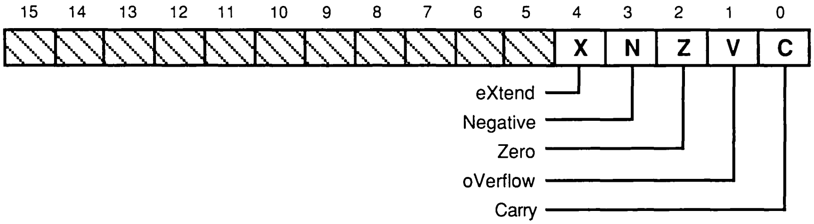


Figure 2-6. Condition Code Register (CCR)

User mode programs cannot modify the status register on any chip.

A diagram of the 68020 status register appears in Figure 2-7. Note that the shading corresponds to hexadecimal digit breakdown.

The upper byte of this register contains information that should not be modified by application programs. Bits 15–14 control instruction tracing, which allows debugging software to run single instructions in software being debugged. Bit 13 determines the processor mode, either supervisor or user. When the processor is in supervisor mode, bit 12 determines which stack pointer (MSP or ISP) is used when A7 is referenced. Bits 11–8 constitute the *interrupt mask*, which determines whether or not the processor will recognize external interrupts. Bits 14 and 12 are not implemented on the 68000 and 68010 chips.

Processor Modes

When bit 13 in the status register is set, the processor is executing in *supervisor*, or *privileged*, mode. In this state, the processor may execute any instruction. When bit 13 is zero, the processor executes in *user* mode. Instructions that can interfere with normal functioning of an operating system cannot be executed in user mode. For instance, all instructions that alter the upper byte of the status register are forbidden to user mode programs (or else they could become supervisor mode programs).

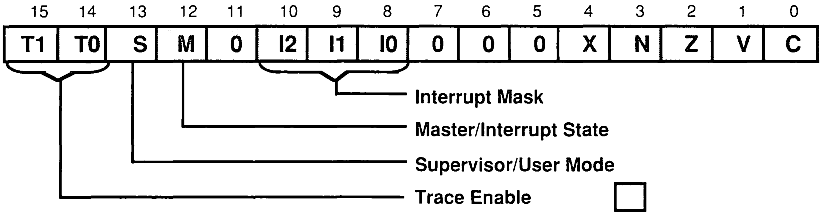


Figure 2-7. Status Register

MSP and ISP

Supervisor mode and user mode programs use different stack pointer registers. When a supervisor mode program references register A7, the 68020 actually references one of two registers: MSP (Master Stack Pointer) or ISP (Interrupt Stack Pointer). The 68010 and 68000 have only one such register, the SSP (Supervisor Stack Pointer). SSP on these older CPUs is equivalent to ISP on the 68020 and subsequent processors. The stack pointer in use is determined by bits 13 (S) and 12 (M) in the status register:

S	**M**	**Stack pointer**
0	0	User Stack Pointer (USP)
0	1	(Illegal combination)
1	0	Interrupt Stack Pointer (ISP)
1	1	Master Stack Pointer (MSP)

The Master Stack Pointer (MSP) is intended for use by normal operating system software. The Interrupt Stack Pointer (ISP) used by interrupt service routines. Many operating systems on the 68000 and 68010 change stack areas (in software) on receipt of an interrupt. The 68020 moves this function into hardware, thus permitting faster interrupt response. User mode A7 is also called the User Stack Pointer (USP). All 68000 chips have instructions that allow supervisor mode software to fetch and set the user stack pointer.

VBR

The interrupt mechanism of the 68000 involves a table with the addresses of routines that handle specific interrupts. These addresses are called *vectors*, and the table is called the *vector table*. The 68010 and subsequent processors allow the vector table to begin at any even address in memory, using a register called the *Vector Base Register* to specify the address. Since the 68000 did not have this register, the vector table was forced to begin at address zero. The interrupt mechanism is described in a later section of this chapter.

CAAR and CACR

The 68020 has two registers that control the operation of the instruction cache. These are the *CAche Address Register (CAAR)* and *CAche Control Register (CACR)*. These registers are described in the section on cache operation later in this chapter.

SFC and DFC

All 68000 chips have three pins that are used to indicate additional information about memory accesses. These function code pins, called FC0, FC1, and FC2, indicate one of eight possible states associated with a memory request:

FC2	FC1	FC0	Interpretation
0	0	0	Reserved for future definition by Motorola
0	0	1	User data space
0	1	0	User program space
0	1	1	Reserved for future definition by system designers
1	0	0	Reserved for future definition by Motorola
1	0	1	Supervisor data space
1	1	0	Supervisor program space
1	1	1	CPU space

FC2–FC0 can potentially be used as extra address lines, placing user data, user code, supervisor data, and supervisor code in separate 4Gb memory areas. This leads to the interesting question of how a supervisor program can load a user program, or exchange data with a user program. Processors starting with the 68010 have two registers and an instruction that moves data using any desired values of FC2–FC0. The instruction is *MOVES (MOVE address Space)*, and the registers are the *Source Function Code (SFC)* and *Destination Function Code (DFC)* registers. These registers are 3 bits wide and contain the desired FC2–FC0 value for the source (SFC) and destination (DFC) operands of the MOVES instruction.

Addressing Modes

Many 68000 instructions use a generalized technique called *addressing mode* for specifying the location of data to be used by the instruction. An instruction using this technique has 6 bits that specify the addressing mode as shown in Figure 2-8.

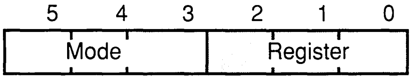


Figure 2-8. 68000 Addressing Mode

The addressing mode specifies a register and a means of using the register (the mode field). Some addressing modes do not involve a register, but simply use all 6 bits to specify the operand's location. Addressing modes are commonly found in the low-order 6 bits of the first word of an instruction. Certain modes require additional data following the instruction. The addressing modes are as follows:

Mode	Register	Interpretation
0	Data Register	Data register contains operand
1	Address Register	Address register contains operand
2	Address Register	Register contains address of operand
3	Address Register	Register contains address of operand. Incremented after use.
4	Address Register	Register contains address of operand. Decremented before use.
5	Address Register	Register + displacement field is address of operand
6	Address Register	Register + displacement (+ index) is address of operand
7	0	Next word (16 bits) contains address of operand
7	1	Next longword (32 bits) contains address of operand
7	2	Program counter + displacement is address of operand
7	3	Program counter + displacement (+ index) (+ base) is address of operand
7	4	Next word or longword contains operand
7	5	Reserved
7	6	Reserved
7	7	Reserved

Parentheses denote optional items. A detailed description of each addressing mode follows.

Data Register Direct

Addressing mode zero is called *data register direct*, meaning that the operand is contained in a data register. The register field specifies the desired data register, as shown in Figure 2-9.

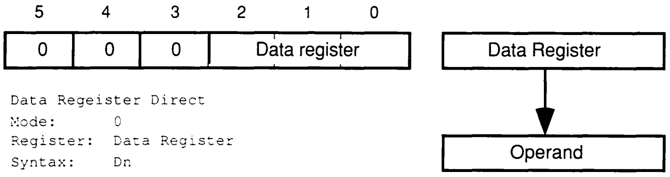

5 4 3 2 1 0
0
0
0
Data register
Data Register
Operand
Data Regeister Direct
Mode: 0
Register: Data Register
Syntax: Dn

Figure 2-9. Data Register Direct Addressing Mode

Many instructions implicitly use this addressing mode by requiring one or more operands to reside in data registers.

Address Register Direct

Addressing mode 1 uses the contents of an address register as the operand. This mode is called *address register direct*. The register field specifies the desired address register, as shown in Figure 2-10.

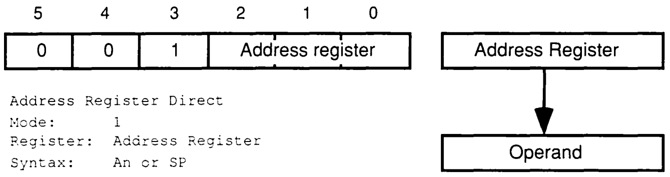
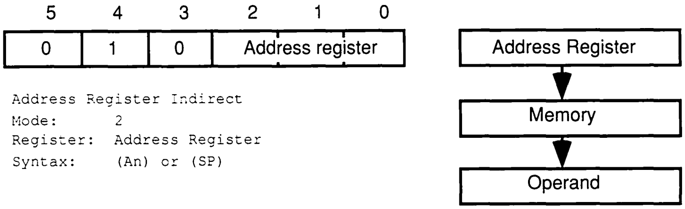

5 4 3 2 1 0
0
0
1
Address register
Address Register
Operand
Address Register Direct
Mode: 1
Register: Address Register
Syntax: An or SP

Figure 2-10. Address Register Direct Addressing Mode

Address Register Indirect

Addressing mode 2 takes the contents of an address register as the memory address of the operand, as shown in Figure 2-11.


5
4
3
2
1
0
0
1
0
Address register
Address Register
Memory
Operand

Address Register Indirect
Mode: 2
Register: Address Register
Syntax: (An) or (SP)

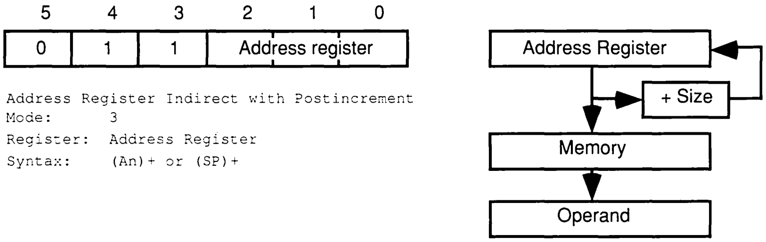
Figure 2-11. Address Register Indirect Addressing Mode

On the 68000 and 68010, a word or longword reference requires that the address register contain an even number. This restriction does not apply on the 68020 and 68030.

Address Register Indirect with Postincrement

Addressing mode 3 uses the contents of an address register as the memory address of the operand. The operand size (1, 2, or 4) is added to the register after use.

This addressing mode is useful for stepping through sequential data items in memory. Postincrement mode is also used for removing items from a stack (also called a *pop* operation). When A7 (or SP) is the address register in a byte operation with this mode, the increment is by 2, thereby forcing the stack address to remain even.


5
4
3
2
1
0
0
1
1
Address register
Address Register
+ Size
Memory
Operand

Address Register Indirect with Postincrement
Mode: 3
Register: Address Register
Syntax: (An)+ or (SP)+

Figure 2-12. Address Register Indirect With Postincrement

Address Register Indirect With Predecrement

Addressing mode 4 uses a predecrement adjustment to an address register before the register is used as the address of the operand in memory.

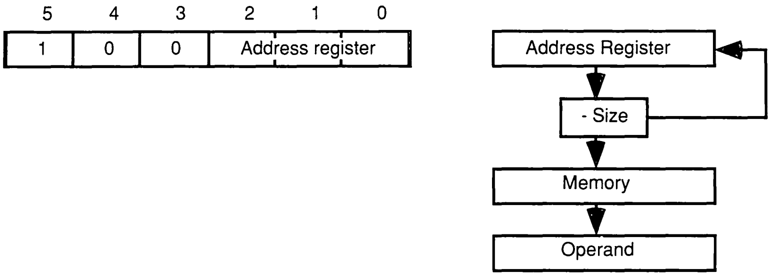
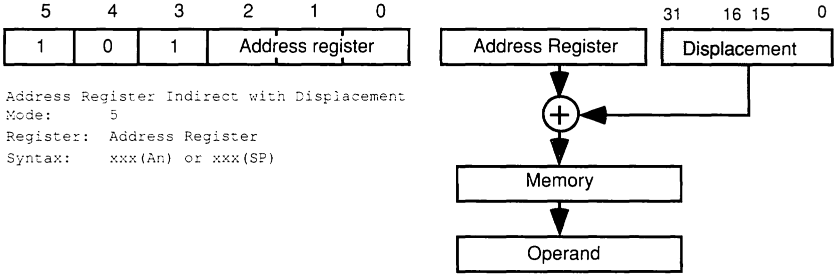

5 4 3 2 1 0
1 0 0 Address register
Address Register
- Size
Memory
Operand

Figure 2-13. Address Register Indirect With Predecrement

Predecrement mode is useful for stepping through items in memory in reverse order. This addressing mode is useful for performing multiple precision adds, subtracts, and other arithmetic operations, since data is stored with the least significant portion (which must be used first) at the highest address. Predecrement mode is also used to add a data item to a stack (also called a *push* operation). When using the stack pointer (A7) with byte-sized data, the register is decremented by two forcing the stack address to remain even.

Address Register Indirect With Displacement

Addressing mode 5 takes the contents of an address register, adds a 16-bit quantity (sign-extended to 32 bits), and uses the result as the memory address of the operand.


5 4 3 2 1 0
1 0 1 Address register
Address Register
31 16 15 0
Displacement
+
Memory
Operand

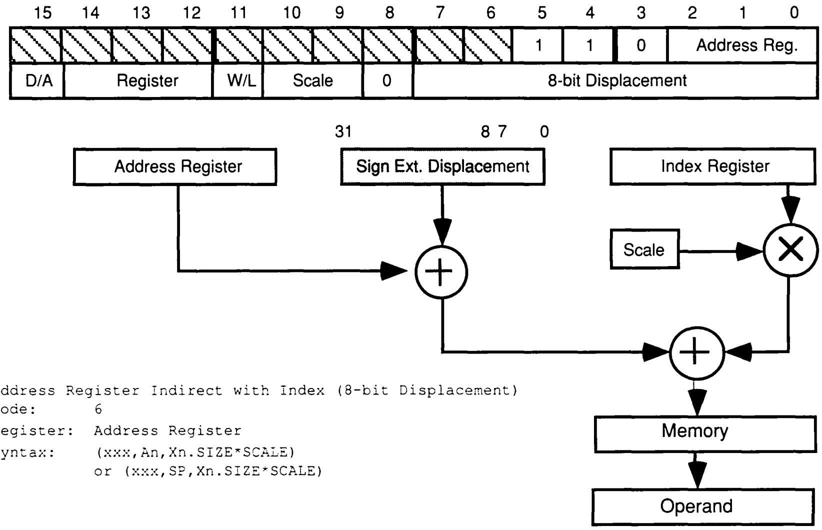
```
Address Register Indirect with Displacement
Mode:      5
Register:  Address Register
Syntax:    xxx(An) or xxx(SP)
```

Figure 2-14. Address Register Indirect With Displacement

Sign extension allows a single address register to serve as the base address for two data areas, each of which must be less than 32,767 (32K) bytes in length. This technique is used in Macintosh applications for the segment table (positive offsets from A5) and application global variables (negative offsets from A5).

Address Register Indirect With Index (8-bit Displacement)

Addressing mode 6 takes from one to five extension words. Depending on the content of these extension words, one of four addressing modes is selected. The first of these combines an address register, an index register, and an 8-bit displacement to generate a memory address.



```
Address Register Indirect with Index (8-bit Displacement)
Mode:       6
Register:   Address Register
Syntax:     (xxx,An,Xn.SIZE*SCALE)
            or (xxx,SP,Xn.SIZE*SCALE)
```

Figure 2-15. Address Register Indirect With Index (8-bit displacement)

The 8-bit displacement field of the second word of the instruction is extended to 32 bits. (Bit 7 is replicated in bits 8-31.) This quantity plus the address register plus the index register (times the scale factor) forms the memory address of the operand.

The second word of the instruction is called the *extension word.* Bit 8 of this word, when zero, indicates the 8-bit displacement form shown above. The D/A flag (bit 15 of the extension word) is one if the index register is an address register, and zero if a data register. The register field (bits 14–12) contains the register number. The W/L flag (bit 11) is one if all 32 bits of the index register participate in the calculation, and zero if only the low-order 16 bits (sign-extended to 32) are to be used. You select this option using a size specifier on the index register (for example, D0.W, or A1.L).

The index register scale factor resides in bits 10–9 of the extension word. This quantity selects the number of bits to shift the index register, resulting in a multiplication by a power of two. The encodings are as follows:

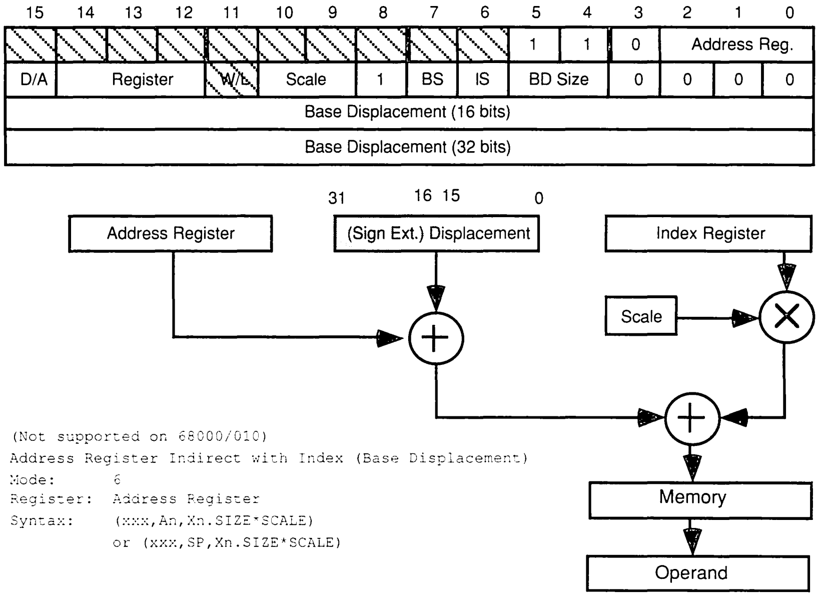
Bit 10	Bit 9	Resulting Index value
0	0	Contents of the index register (unscaled)
0	1	Contents of the index register times 2
1	0	Contents of the index register times 4
1	1	Contents of the index register times 8

The 68000 and 68010 do not support index scaling. Bits 10–9 are ignored by these processors, and the unscaled index register participates in the address calculation.

[68020/30 only]

Address Register Indirect With Index (Base Displacement)

A second variation on mode 6 uses an address register, a scaled index register, and an optional 16 or 32-bit displacement. All of these components are optional. If all components are omitted, the effective address is zero. (Note that the extension word may not be omitted.)


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1
1
0
Address Reg.
D/A
Register
W/L
Scale
1
BS
IS
BD Size
0
0
0
0
Base Displacement (16 bits)
Base Displacement (32 bits)
31 16 15 0
Address Register
(Sign Ext.) Displacement
Index Register
Scale
Memory
Operand

```
(Not supported on 68000/010)
Address Register Indirect with Index (Base Displacement)
Mode:      6
Register:  Address Register
Syntax:    (xxx,An,Xn.SIZE*SCALE)
           or (xxx,SP,Xn.SIZE*SCALE)
```

Figure 2-16. Address Register Indirect With Index (Base displacement)

The second word of the instruction is called the *extension word.* Bit 8 of this word, when one, indicates the base displacement form shown above. The D/A flag (bit 15 of the extension word) is one if the index register is an address register, and zero if a data register. The register field (bits 14–12) contains the register number. The W/L flag (bit 11) is one if all 32 bits of the index register participate in the calculation and zero if only the low-order 16 bits (sign-extended to 32) are to be used. You select this option using a size specifier on the index register (for example, D0.W, or A1.L).

The index register scale factor resides in bits 10–9 of the extension word. This quantity selects the number of bits to shift the index register, resulting in a multiplication by a power of two. The encodings are as follows:

Bit 10	Bit 9	Resulting Index value
0	0	Contents of the index register
0	1	Contents of the index register times 2
1	0	Contents of the index register times 4
1	1	Contents of the index register times 8

The 68000 and 68010 do not support index scaling. Bits 10–9 are ignored by these processors, and the unscaled index register participates in the address calculation.

Bit 7 of the extension word (BS) is the Base Suppress flag. When this bit is set to one, the base register (the address register in Figure 2-16) does not participate in the address calculation. You specify this option by using a *zero suppressed address register* specification to the assembler. Register names ZA0 though ZA7 cause this bit to be set. The use of this feature allows you to use a data register (when specified as the index) as an address register.

Bit 6 of the extension word (IS) is the Index Suppress flag. When this bit is set to one, the index register (specified in bits 15–12 of the extension word) is replaced by a constant value of zero in the address calculation. This feature is useful when you need a 32-bit displacement from an address register.

Bits 4–5 (BD Size) give the number of words of base displacement that follow the extension word.

Bit 5	Bit 4	Resulting Base Displacement Size
0	0	(Illegal combination)
0	1	No base displacement field (value is zero)
1	0	Base displacement is 16 bits (sign-extended to 32)
1	1	Base displacement is 32 bits

[68020/30 only] *Memory Indirect Post-Indexed*

Another option available with addressing mode 6 is to use a pointer in memory as part of operand address generation. The address register and base displacement point to a longword in memory. The contents of this longword, plus an optional scaled index and/or an optional 32-bit constant (called the *outer displacement*), form the address of the operand. This process is illustrated in Figure 2-17.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	1	1	0	Address Reg.		
D/A	Register			W/L	Scale		1	BS	IS	BD Size		0	0	OD Size	
Base Displacement (16 bits)															
Base Displacement (32 bits)															
Outer Displacement (16 bits)															
Outer Displacement (32 bits)															

```
(Not supported on 68000/010)
Memory Indirect Post-Indexed
Mode:       6
Register:   Address Register
Syntax:      ([bd,An],Xn.SIZE*SCALE,od)
          or ([bd,SP],Xn.SIZE*SCALE,od)
```

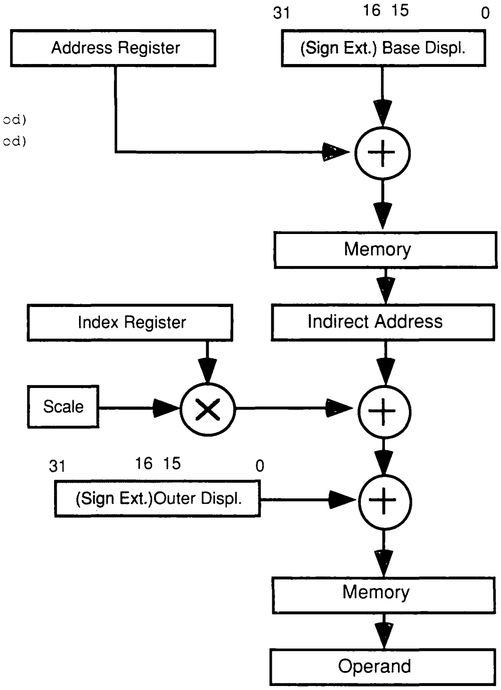

31
16 15
0
Address Register
(Sign Ext.) Base Displ.
Memory
Index Register
Indirect Address
Scale
31
16 15
0
(Sign Ext.)Outer Displ.
Memory
Operand

Figure 2-17. Memory Indirect Post-Indexed

The D/A flag (bit 15 of the extension word) is one if the index register is an address register and zero if a data register. The register field (bits 14–12) contains the register number. The W/L flag (bit 11) is one if all 32 bits of the index register participate in the calculation, and zero if only the low-order 16 bits (sign-extended to 32) are to be used. You select this option using a size specifier on the index register (for example, D0.W, or A1.L).

The index register scale factor resides in bits 10–9 of the extension word. This quantity selects the number of bits to shift the index register, resulting in a multiplication by a power of two. The encodings are as follows:

Bit 10	Bit 9	Resulting Index value
0	0	Contents of the index register
0	1	Contents of the index register times 2
1	0	Contents of the index register times 4
1	1	Contents of the index register times 8

The 68000 and 68010 do not support index scaling. Bits 10–9 are ignored by these processors, and the unscaled index register participates in the address calculation.

Bit 7 of the extension word (BS) is the Base Suppress flag. When this bit is set to one, the base register (the address register in Figure 2-17) does not participate in the address calculation. You specify this option by using a *zero suppressed address register* specification to the assembler. Register names ZA0 though ZA7 cause this bit to be set. Note that the use of this feature allows you to use a data register (when specified as the index) as an address register.

Bit 6 of the extension word (IS) is the Index Suppress flag. When this bit is set to one, the index register (specified in bits 15–12 of the extension word) is replaced by a constant value of zero in the address calculation. This feature is useful when you need a 32-bit displacement from an address register.

Bits 5–4 (BD Size) give the number of words of base displacement that follow the extension word.

Bit 5	Bit 4	Resulting Base Displacement Size
0	0	(Illegal combination)
0	1	No base displacement field (value is zero)
1	0	Base displacement is 16 bits (sign-extended to 32)
1	1	Base displacement is 32 bits

Bits 1–0 (OD Size) give the number of words for the outer displacement.

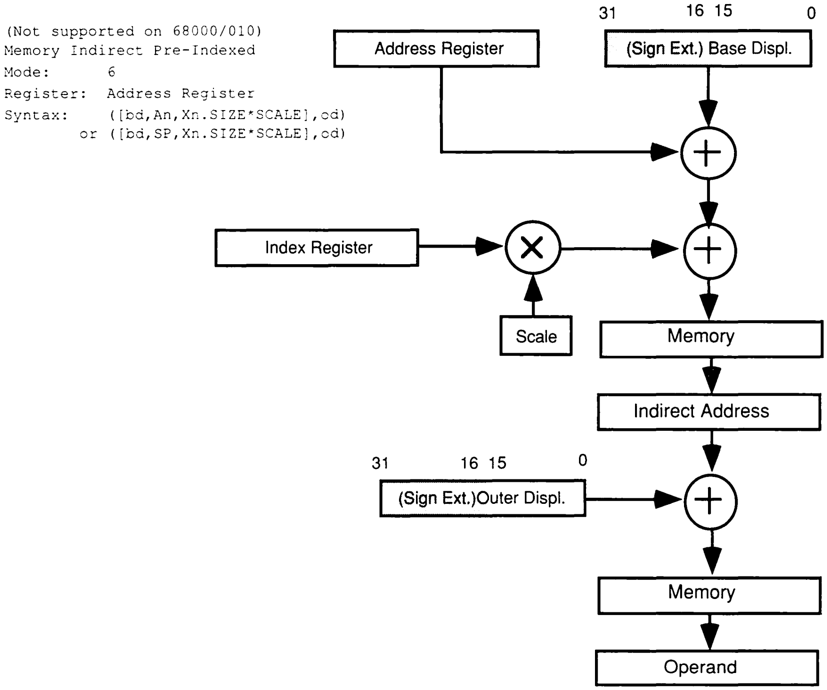
Bit 1	Bit 0	Resulting Outer Displacement Size
0	0	(Illegal combination)
0	1	No outer displacement field (value is zero)
1	0	Outer displacement is 16 bits (sign-extended to 32)
1	1	Outer displacement is 32 bits

[68020/30 only] *Memory Indirect Pre-Indexed*

The indirect pre-indexed addressing mode is very similar to the indirect post-indexed mode, except that the index register participates in the indirect address calculation rather than the operand address calculation. This mode is shown in Figure 2-18.

The extension word fields are as described for the post-indexed addressing mode.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	1	1	0	Address Reg.		
D/A	Register			W/L	Scale		1	BS	IS	BD Size		0	1	OD Size	
Base Displacement (16 bits)															
Base Displacement (32 bits)															
Outer Displacement (16 bits)															
Outer Displacement (32 bits)															



```
(Not supported on 68000/010)
Memory Indirect Pre-Indexed
Mode:       6
Register:   Address Register
Syntax:     ([bd,An,Xn.SIZE*SCALE],od)
         or ([bd,SP,Xn.SIZE*SCALE],od)
```

Address Register
31
16 15
0
(Sign Ext.) Base Displ.
Index Register
Scale
Memory
Indirect Address
31
16 15
0
(Sign Ext.)Outer Displ.
Memory
Operand

Figure 2-18. **Memory Indirect Pre-Indexed**

Program Counter Indirect with Displacement

This addressing mode uses the program counter plus a sign-extended 16-bit displacement as the operand memory address. The program counter value is the address of the second word of the instruction (the address of the 16-bit displacement).

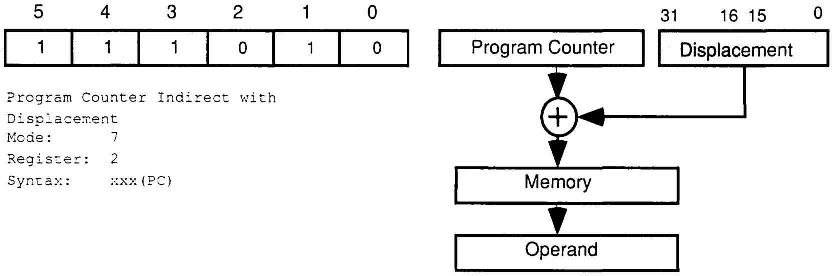


Figure 2-19. Program Counter Indirect with Displacement

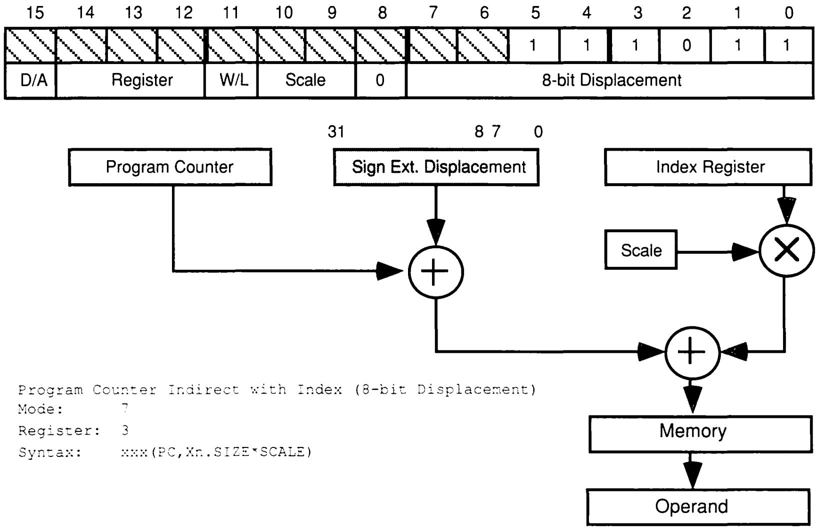
This mode is similar to Address Register Indirect with Displacement (mode 5) with two exceptions: program counter addressing modes may not be used as destination operands, and the FC2–FC0 bits reflect code access rather than data access.

Program Counter Indirect with Index (8-Bit Displacement)

You can also combine the program counter, an index register, and an 8-bit sign-extended displacement to form the operand address. All 32 bits or the low 16 bits (sign-extended) of the index register can be used.

The 8-bit displacement field of the second word of the instruction is extended to 32 bits (bit 7 is replicated in bits 8-31). This quantity plus the address register plus the index register (times the scale factor) forms the memory address of the operand.

The second word of the instruction is called the *extension word.* Bit 8 of this word, when zero, indicates the 8-bit displacement form shown above. The D/A flag (bit 15 of the extension word) is one if the index register is an address register, and zero if a data register.


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 1 1
D/A
Register
W/L
Scale
0
8-bit Displacement
31 8 7 0
Program Counter
Sign Ext. Displacement
Index Register
Scale
Memory
Operand

Program Counter Indirect with Index (8-bit Displacement)
Mode: 7
Register: 3
Syntax: xxx(PC,Xn.SIZE*SCALE)

Figure 2-20. Program Counter Indirect with Index (8-bit Displacement)

The register field (bits 14–12) contains the register number. The W/L flag (bit 11) is one if all 32 bits of the index register participate in the calculation, and zero if only the low-order 16 bits (sign-extended to 32) are to be used. You select this option using a size specifier on the index register (for example, D0.W, or A1.L). The value of the program counter used in the address computation is the address of the extension word.

The index register scale factor resides in bits 10–9 of the extension word. This quantity selects the number of bits to shift the index register, resulting in a multiplication by a power of two. The encodings are as follows:

Bit 10	**Bit 9**	**Resulting Index value**
0	0	Contents of the index register (unscaled)
0	1	Contents of the index register times 2
1	0	Contents of the index register times 4
1	1	Contents of the index register times 8

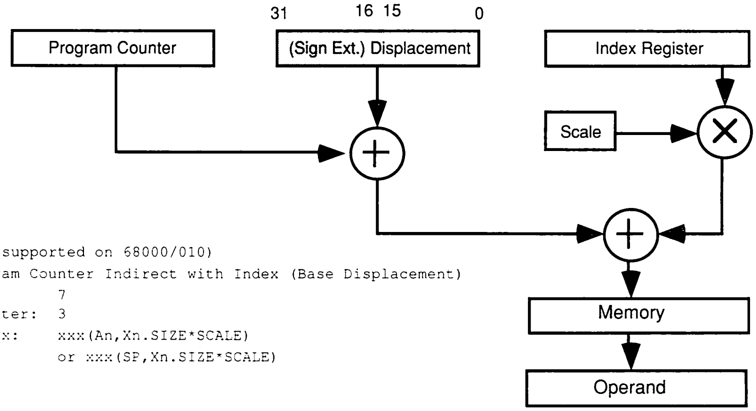
The 68000 and 68010 do not support index scaling. Bits 10–9 are ignored by these processors, and the unscaled index register participates in the address calculation.

Program counter relative addressing modes may not be used for destination operands. The function code bits (FC0–FC2) reflect a code access for this addressing mode.

[68020/30 only] *Program Counter Indirect With Index (Base Displacement)*

This addressing mode uses the program counter, a scaled index register, and an optional 16- or 32-bit displacement. All of these components are optional. If all components are omitted, the effective address is zero. (Note that the extension word may not be omitted.)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	1	1	1	0	1	1
D/A	Register			W/L	Scale		1	BS	IS	BD Size		0	0	0	0
Base Displacement (16 bits)															
Base Displacement (32 bits)															



```
(Not supported on 68000/010)
Program Counter Indirect with Index (Base Displacement)
Mode:      7
Register:  3
Syntax:    xxx(An,Xn.SIZE*SCALE)
           or xxx(SP,Xn.SIZE*SCALE)
```

Figure 2-21. Program Counter Indirect with Index (Base Displacement)

The second word of the instruction is called the *extension word.* Bit 8 of this word, when one, indicates the base displacement form shown above. The D/A flag (bit 15 of the extension word) is one if the index register is an address register, and zero if a data register. The register field (bits 14–12) contains the register number. The W/L flag (bit 11) is one if all 32 bits of the index register participate in the calculation, and zero if only the low-order 16 bits (sign-extended to 32) are to be used. You select this option using a size specifier on the index register (for example, D0.W, or A1.L). The value of the program counter used in the address computation is the address of the extension word.

The index register scale factor resides in bits 10–9 of the extension word. This quantity selects the number of bits to shift the index register, resulting in a multiplication by a power of two. The encodings are as follows:

Bit 10	**Bit 9**	**Resulting Index value**
0	0	Contents of the index register
0	1	Contents of the index register times 2
1	0	Contents of the index register times 4
1	1	Contents of the index register times 8

The 68000 and 68010 do not support index scaling. Bits 10–9 are ignored by these processors, and the unscaled index register participates in the address calculation.

Bit 7 of the extension word (BS) is the Base Suppress flag. When this bit is set to one, the program counter does not participate in the address calculation. You specify this option by using the *zero suppressed PC* (ZPC) specification to the assembler. Note that the use of this feature allows you to use a data register (when specified as the index) as an address register.

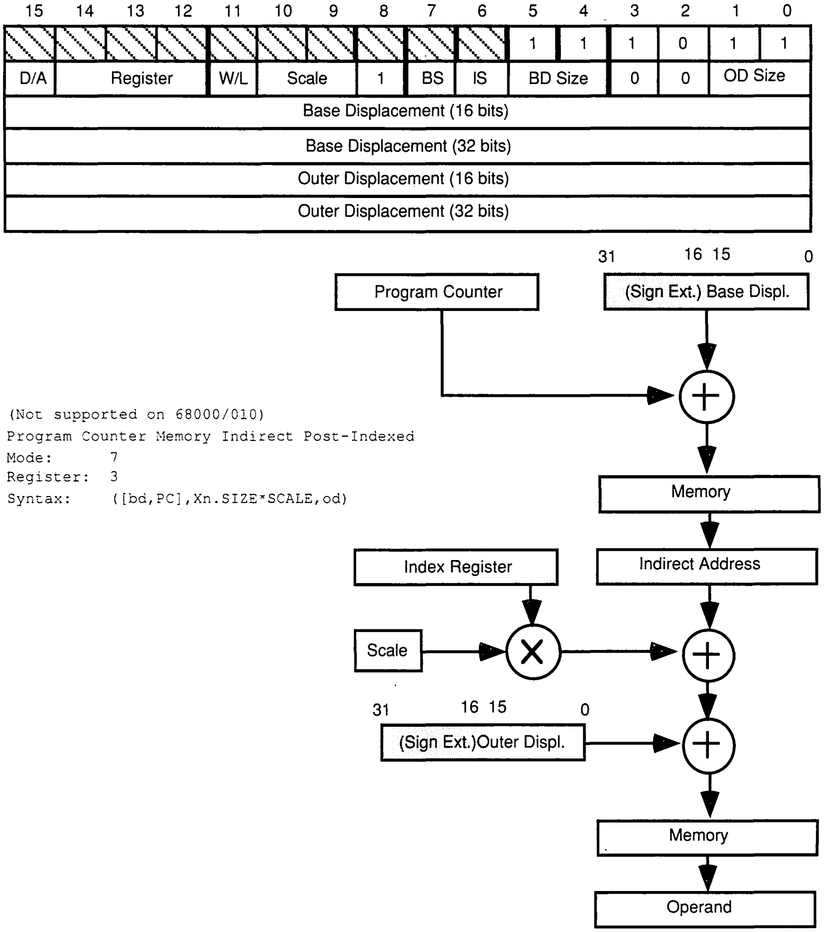
Bit 6 of the extension word (IS) is the Index Suppress flag. When this bit is set to one, the index register (specified in bits 15–12 of the extension word) is replaced by a constant value of zero in the address calculation. This feature is useful when you need a 32-bit displacement from an address register.

Bits 4–5 (BD Size) give the number of words of base displacement that follow the extension word.

Bit 5	**Bit 4**	**Resulting Base Displacement Size**
0	0	(Illegal combination)
0	1	No base displacement field (value is zero)
1	0	Base displacement is 16 bits (sign-extended to 32)
1	1	Base displacement is 32 bits

[68020/30 only] *Program Counter Memory Indirect Post-Indexed*

Another option available with PC-relative addressing is to use a pointer in memory as part of operand address generation. The address register and base displacement point to a longword in memory. The contents of this longword, plus an optional scaled index and/or an optional 32-bit constant (called the *outer displacement*) form the address of the operand. This process is illustrated in Figure 2-22.


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 1 1
D/A
Register
W/L
Scale
1
BS
IS
BD Size
0
0
OD Size
Base Displacement (16 bits)
Base Displacement (32 bits)
Outer Displacement (16 bits)
Outer Displacement (32 bits)
31 16 15 0
Program Counter
(Sign Ext.) Base Displ.
Memory
Index Register
Indirect Address
Scale
31 16 15 0
(Sign Ext.)Outer Displ.
Memory
Operand

```
(Not supported on 68000/010)
Program Counter Memory Indirect Post-Indexed
Mode:      7
Register:  3
Syntax:    ([bd,PC],Xn.SIZE*SCALE,od)
```

Figure 2-22. Program Counter Memory Indirect Post-Indexed

The D/A flag (bit 15 of the extension word) is one if the index register is an address register, and zero if a data register. The register field (bits 14–12) contains the register number. The W/L flag (bit 11) is one if all 32 bits of the index register participate in the calculation, and zero if only the low-order 16 bits (sign-extended to 32) are to be used.

You select this option using a size specifier on the index register (for example, D0.W, or A1.L). The value of the program counter used in the address computation is the address of the extension word.

The index register scale factor resides in bits 10–9 of the extension word. This quantity selects the number of bits to shift the index register, resulting in a multiplication by a power of two. The encodings are as follows:

Bit 10	**Bit 9**	**Resulting Index value**
0	0	Contents of the index register
0	1	Contents of the index register times 2
1	0	Contents of the index register times 4
1	1	Contents of the index register times 8

The 68000 and 68010 do not support index scaling. Bits 10–9 are ignored by these processors, and the unscaled index register participates in the address calculation.

Bit 7 of the extension word (BS) is the Base Suppress flag. When this bit is set to one, the program counter does not participate in the address calculation. You specify this option by using the *zero suppressed PC* (ZPC) specification to the assembler. This feature allows you to use a data register (when specified as the index) as an address register.

Bit 6 of the extension word (IS) is the Index Suppress flag. When this bit is set to one, the index register (specified in bits 15–12 of the extension word) is replaced by a constant value of zero in the address calculation. This feature is useful when you need a 32-bit displacement from an address register.

Bits 5–4 (BD Size) give the number of words of base displacement that follow the extension word.

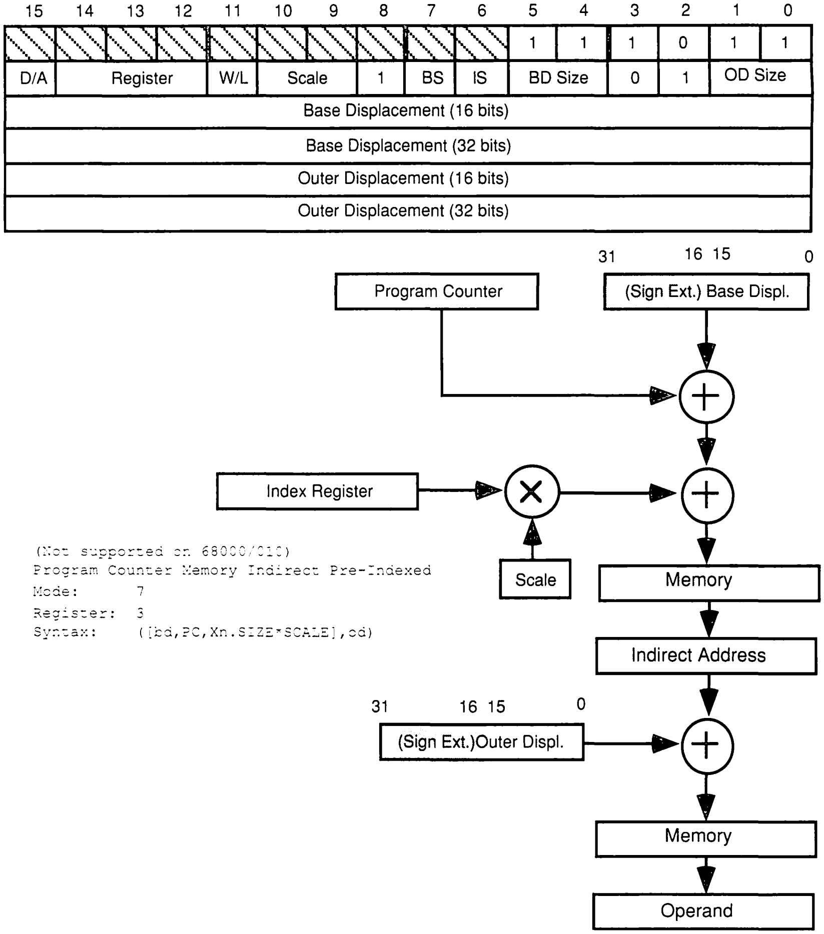
Bit 5	**Bit 4**	**Resulting Base Displacement Size**
0	0	(Illegal combination)
0	1	No base displacement field (value is zero)
1	0	Base displacement is 16 bits (sign-extended to 32)
1	1	Base displacement is 32 bits

Bits 1–0 (OD Size) give the number of words for the outer displacement.

Bit 1	**Bit 0**	**Resulting Outer Displacement Size**
0	0	(Illegal combination)
0	1	No outer displacement field (value is zero)
1	0	Outer displacement is 16 bits (sign-extended to 32)
1	1	Outer displacement is 32 bits

[68020/30 only] *Program Counter Memory Indirect Pre-Indexed*

The PC-relative indirect pre-indexed addressing mode is very similar to the indirect post-indexed mode, except that the index register participates in the indirect address calculation rather than the operand address calculation. This mode is shown in Figure 2-23.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	1	1	1	0	1	1
D/A	Register			W/L	Scale		1	BS	IS	BD Size		0	1	OD Size	
Base Displacement (16 bits)															
Base Displacement (32 bits)															
Outer Displacement (16 bits)															
Outer Displacement (32 bits)															

31
16 15
0
Program Counter
(Sign Ext.) Base Displ.
Index Register
Scale
Memory
Indirect Address
31
16 15
0
(Sign Ext.)Outer Displ.
Memory
Operand

```
(Not supported on 68000/010)
Program Counter Memory Indirect Pre-Indexed
Mode:       7
Register:   3
Syntax:     ([bd,PC,Xn.SIZE*SCALE],od)
```

Figure 2-23. Program Counter Memory Indirect Pre-Indexed

The extension word fields are as described for the Program Counter Memory Indirect Post-Indexed addressing mode.

Absolute Short Address

Addressing mode 7 with the register field set to 0 causes the 16-bit quantity following the instruction to be used as the address of the operand. This 16-bit word is sign-extended to a 32-bit longword before use. Addresses $0–$7FFF and $FFFF8000–$FFFFFFFF can be reached using this addressing mode.

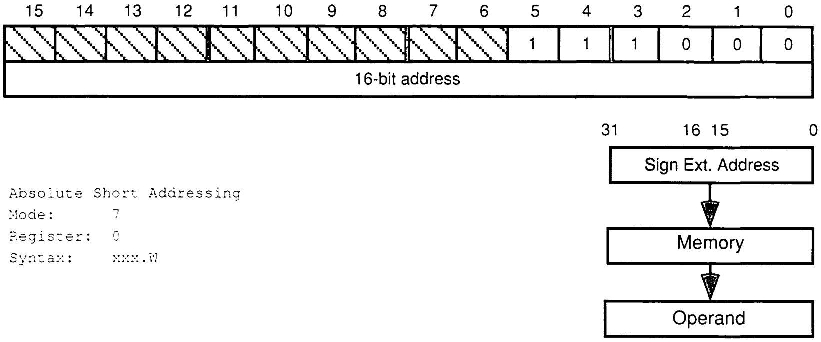

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 0
16-bit address
31 16 15 0
Sign Ext. Address
Memory
Operand
Absolute Short Addressing
Mode: 7
Register: 0
Syntax: xxx.W

Figure 2-24. Absolute Short Addressing

Some systems (notably Macintosh) have global data in the low 32K ($0–$7FFF) of memory. This addressing mode is useful for fetching and changing this data.

Absolute Long Addressing

Mode 7 with the register field set to 1 uses the next 32 bits following the instruction as the address of the operand.

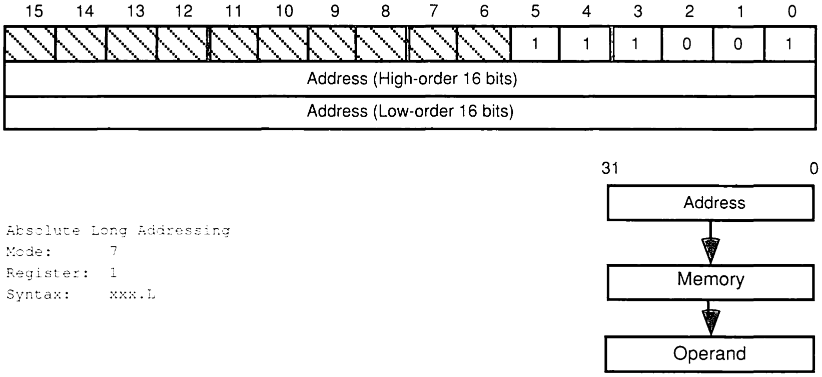
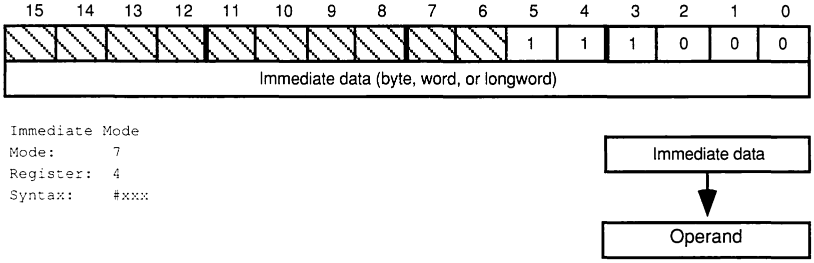

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1
Address (High-order 16 bits)
Address (Low-order 16 bits)
31 0
Address
Memory
Operand
Absolute Long Addressing
Mode: 7
Register: 1
Syntax: xxx.L

Figure 2-25. Absolute Long Addressing

This addressing mode is widely used in software that runs on the 68000 and 68010 chips because it allows arbitrary sized programs and data areas. Its disadvantage is that the addresses must reflect the program load address. This mode is not supported on the Macintosh, and programs for that system must be split into 32K segments. The 68020 PC-relative addressing modes remove the 16-bit restriction, thus eliminating the segmentation requirement.

Immediate Data

Mode 7 with the register field set to 4 causes the byte, word, or longword following the instruction to be used as the operand. This is called *immediate mode*, and is valid only for source operands.



Immediate Mode
Mode: 7
Register: 4
Syntax: #xxx

Figure 2-26. Immediate Mode

Memory Cache Operation

The 68020 and 68030 have special on-chip memory known as *cache memory*. The idea behind cache memory is to keep data that is likely to be used soon inside the chip itself so that fewer memory accesses are required. Accessing cache memory is much faster than accessing main memory. A good caching scheme can significantly improve execution speed.

There are two types of cache memory in widespread use today: *instruction cache* and *data cache*. Instruction caching means that only memory fetched as instructions participates in caching. The 68020 has only an instruction cache. A data cache attempts to keep memory likely to be referenced as data in cache as well. The 68030 has both instruction and data caches.

68020 Cache

The 68020 has 256 bytes (64 longwords) of on-board cache memory. This memory is used to cache instructions only. Figure 2-27 shows the 68020 cache architecture.

The cache consists of two pieces: an array of *tags* and an array of *data*. The data is a copy of a memory longword, identified by the function codes FC2–FC0 and a 32-bit address. There are 64 longwords of data in the cache. Address bits A7–A2 select the data item, and the associated tag indicates whether the cache contains a valid copy of the selected address. The tag contains a copy of address bits A31–A8, function codes FC2–FC0 and a bit indicating whether the data entry contains valid information.

When the 68020 fetches an instruction, instruction address bits A7–A2 select the cache entry. If bits A31–A8 and FC2–FC0 in the associated cache tag match the requested address, and the entry is marked valid, the processor uses the word selected by A1 from the cache data entry. This is known as a *cache hit*. A hit allows the processor to re-fetch a previously accessed instruction without accessing memory. For program loops that fit in cache, the processor accesses memory only to fetch and store data. For a cache miss, the 68020 fetches the longword containing the instruction and places this data in the appropriate cache entry, updating the tag field as well.

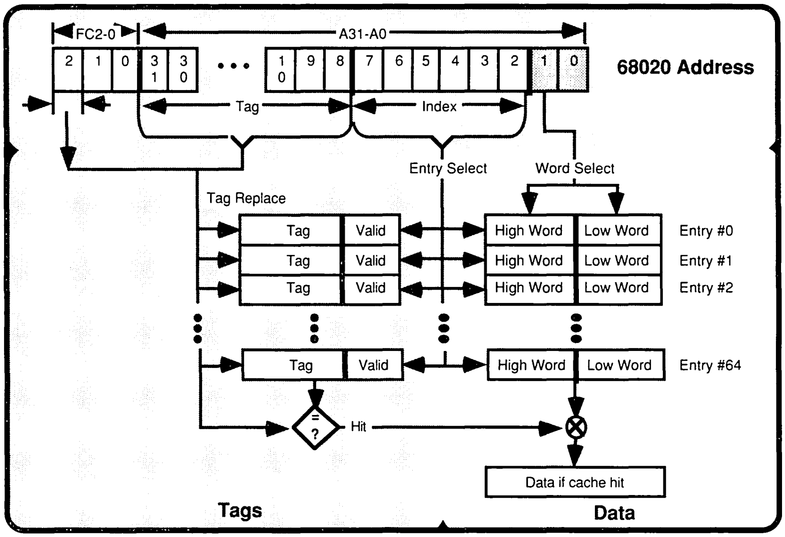

FC2-0
A31-A0
2
1
0
31
30
...
10
9
8
7
6
5
4
3
2
1
0
68020 Address
Tag
Index
Entry Select
Word Select
Tag Replace
Tag
Valid
High Word
Low Word
Entry #0
Entry #1
Entry #2
Entry #64
=?
Hit
Data if cache hit
Tags
Data

Figure 2-27. 68020 Cache Architecture

The 68020 provides supervisor mode software with four cache control operations. These functions are performed by setting bits in a special purpose register, the CAche Control Register (CACR). This register can be accessed only by the MOVE Control register (MOVEC) instruction, described in Chapter 3.

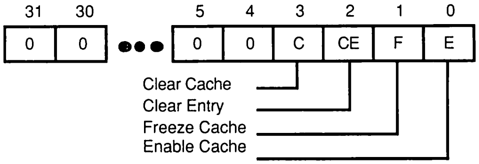


Figure 2-28. 68020 Cache Control Register

Bit 0 of this register is the cache enable bit. Setting this bit to zero forces continuous cache misses and suppresses filling cache entries. The cache is initially disabled when the processor is reset and must be explicitly enabled by software.

CACR bit 1 is the freeze cache bit. Setting this bit to one disables filling cache entries. The cache continues to operate, but no entries are replaced. This bit is normally used by an in-circuit emulator to preserve cache contents during execution of emulator support software. The freeze bit always reads back as zero, and the cache is frozen each time a value with bit 1 set is written to the CACR. Writing a value with bit 1 equal to zero unfreezes the cache.

The Clear Cache bit (bit 3) invalidates all entries in the cache. This function is required in multitasking systems so that cache entries generated by one task are not erroneously used by another task. Software that switches from one task to another sets this bit in the CACR to avoid this problem. There is no need to clear the bit, and it always reads back as zero.

A single cache entry may be invalidated using the Clear Entry bit (bit 2) of the CACR. When this bit is set, the 68020 invalidates an entry in the cache selected by the CAche Address Register (CAAR). This register is shown in Figure 2-29.

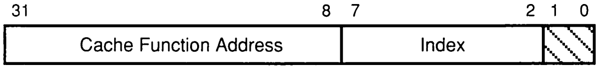


Figure 2-29. Cache Address Register

Only the index field (bits 7–2) is actually used. The other bits are implemented, and Motorola reserves the right to use these bits in later processors. (Don't write any code that uses these bits. You'll regret it later.) The index field selects the cache entry to be invalidated.

68030 Cache

The 68030 has two 256-byte caches, one for instructions and one for data. These caches are organized differently from the 68020. Cache on the 68030 consists of 16 groups of 4 longwords. These groups are called *cache lines.* Whenever the 68030 references memory that is not in cache, the chip fetches the required longword first and then the 3 other longwords that surround it. This technique is known as *burst mode.* A cache line always begins on an address evenly divisible by 16. When fetching a cache line, the 68030 fetches the required data first and the other items in ascending address order. When the highest addressed item is fetched, the 68030 wraps around to the lowest addressed item, as shown in Figure 2-30. Burst mode can be enabled and disabled under software control.

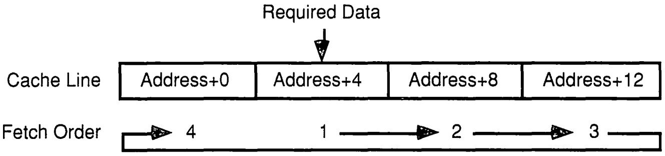


Figure 2-30. Burst Mode Fetch Order

The instruction cache and data cache have slightly different organizations. The instruction cache needs only the supervisor/user bit (FC2) from the function code bits (FC2–FC0), but the data cache needs all three tag bits. Furthermore, the data cache must also accommodate both memory writes and memory reads.

68030 Instruction Cache Figure 2-31 illustrates the organization of the 68030 instruction cache. As with the 68020 cache, the FC2 and A31–A8 address bits comprise the tag. There is one tag per cache line, with a valid bit per longword. Each data

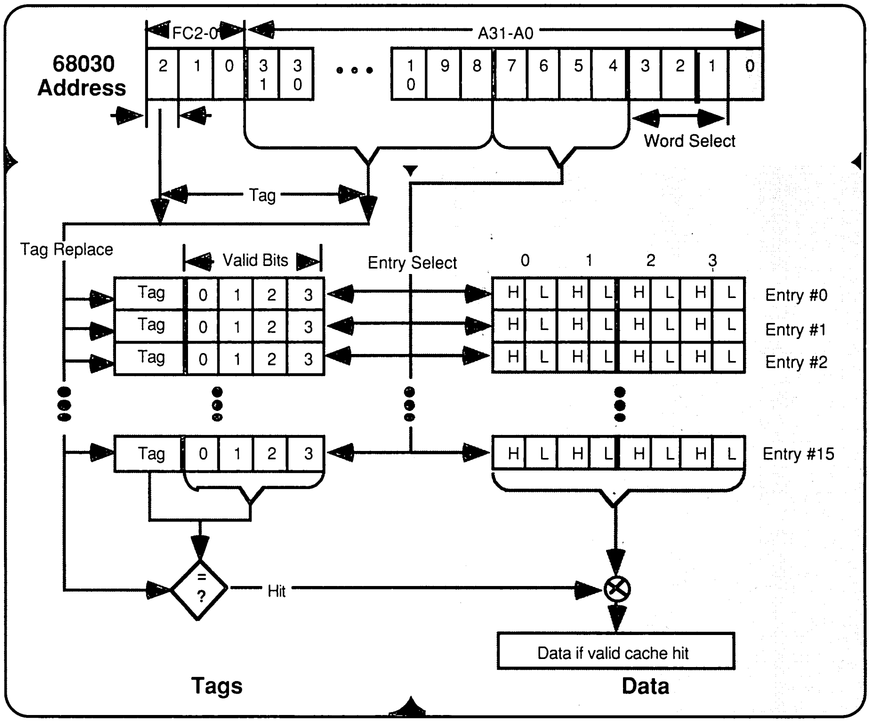

FC2-0
A31-A0
68030 Address
2 1 0 3 1 3 0 ••• 1 0 9 8 7 6 5 4 3 2 1 0
Word Select
Tag
Tag Replace
Valid Bits
Entry Select
0 1 2 3
Tag 0 1 2 3
H L H L H L H L
Entry #0
Entry #1
Entry #2
Entry #15
=?
Hit
Data if valid cache hit
Tags
Data

Figure 2-31. 68030 Instruction Cache

entry consists of 4 longwords. There are 16 such entries. Address bits A7–A4 select the cache entry, and bits A3–A1 select a word within the entry. A cache hit takes place if the tag in cache matches the generated tag and the proper valid bit is set.

68030 Data Cache The 68030 data cache layout is similar to the 68030 instruction cache, except that bits FC1 and FC2 are part of the tag. Bits A3–A2 select a cache longword, and bits A1–A0 select the starting byte within the longword. This layout is shown in Figure 2-32.

Data reads are more difficult than instruction reads. Data reads can be 1, 2, or 4 bytes, as opposed to 2 bytes for instruction fetches. A misaligned word or long fetch can cause replacement of two cache lines. If the first byte of the fetch resides at offset 14 (for longwords) or 15 (for both words and longwords), the selected cache line and the next consecutive cache line will be replaced.

Data writes are more complicated. The 68030 always writes to external memory on a data write, even if the desired data is already in cache. This is called *write-through* operation.

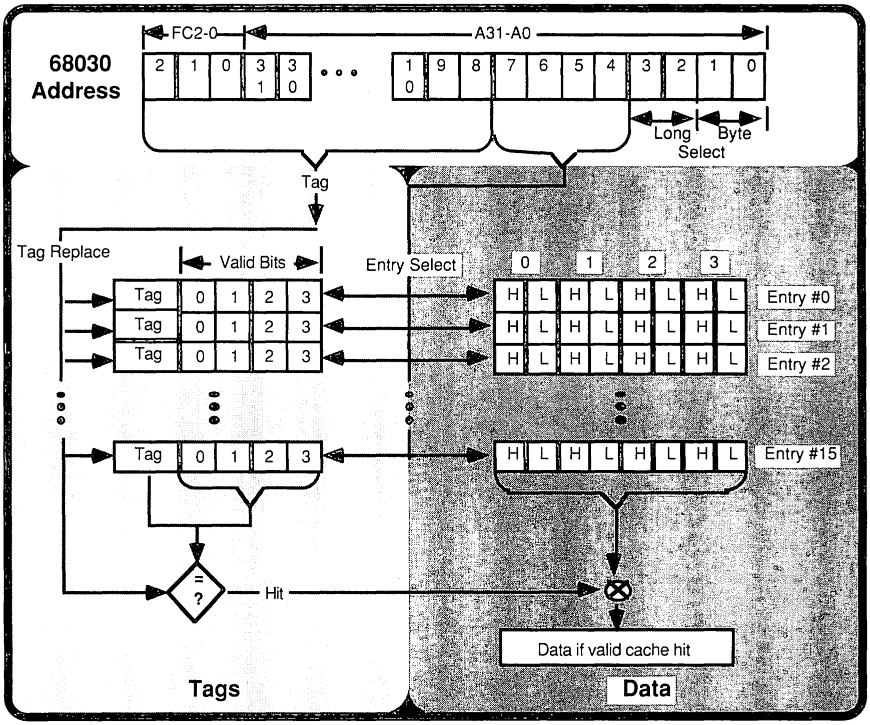

68030 Address
FC2-0
A31-A0
2 1 0 31 30 ... 10 9 8 7 6 5 4 3 2 1 0
Long Select
Byte Select
Tag
Tag Replace
Valid Bits
Entry Select
0 1 2 3
Tag 0 1 2 3
H L H L H L H L
Entry #0
Entry #1
Entry #2
Entry #15
=?
Hit
Data if valid cache hit
Tags
Data

Figure 2-32. 68030 Data Cache

Data writes for addresses not in cache may be handled in one of two ways under software control. The option is called *write allocation*. When write allocation is disabled, the processor does not replace cache entries when a write cycle causes a cache miss. The cache is updated for write hits, however. Thus, a write to a location not in cache will not disturb the selected cache line. The line will still have the last data read from memory (if any). A write to a location that is already in cache (and valid) will cause the data in that cache entry to reflect the written value.

When write allocation is enabled, the processor updates the data cache on cachable writes. A write that causes a cache miss marks the other entries in the cache line as invalid. The longword modified by the write is updated and marked valid if the write is a longword write on a longword boundary. A write that is misaligned or not a longword write marks the cached longword as invalid. The reason for this behavior is that the 68030 does not fill the cache line on writes. Cache misses on non-aligned or non-longword writes would require that the cache entries be read from main memory before being updated. This would mean one or two additional memory cycles.

68030 Cache Control The 68030 has the same two registers as the 68020 for cache control: the CAche Address Register (CAAR) and the CAche Control Register (CACR). Both registers must be accessed using the MOVE Control (MOVEC) instruction. The Cache Address Register is identical to the 68020 CAAR described above. The Cache Control Register is similar to the corresponding register on the 68020, but with some additional fields. This register appears in Figure 2-33.

Bits 13–8 of this register control the 68030 data cache, and bits 4–0 control the instruction cache. Bits 3–0 are compatible with the 68020 CACR. The shaded bits are unique to the 68030. On the 68020, these bits read as zeros and are ignored on writes.

The Write Allocate bit enables the allocation of cache entries on data writes, as described above. Writing a value to the CACR with bit 13 set enables write allocation, and a value with bit 13 clear disables this feature. This bit is cleared when the processor is reset and is ignored if the data cache is disabled or frozen. This bit should be set whenever a supervisor mode program and a user mode program share data or when an MMU is used to map the same area of memory into different logical addresses.

Bit 12 of the CACR is the Data Burst Enable flag. This bit is cleared when the processor is reset. Writing a value to the CACR with bit 12 set allows data cache burst mode filling on data reads. A value with bit 12 clear written to the CACR disables data cache burst mode. When burst mode is disabled, only the required entries are filled. Note that data misalignment may require fetching 2 longwords for a single operand fetch from memory.

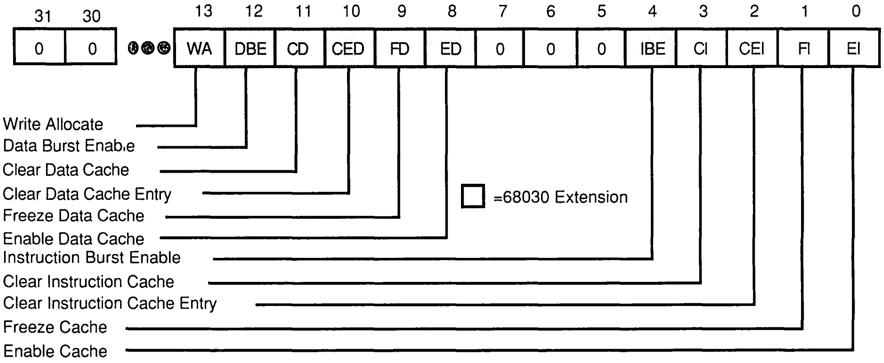

31
30
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0
0
WA
DBE
CD
CED
FD
ED
0
0
0
IBE
CI
CEI
FI
EI
Write Allocate
Data Burst Enable
Clear Data Cache
Clear Data Cache Entry
Freeze Data Cache
Enable Data Cache
Instruction Burst Enable
Clear Instruction Cache
Clear Instruction Cache Entry
Freeze Cache
Enable Cache
☐ =68030 Extension

Figure 2-33. 68030 CAche Control Register (CACR)

To clear the data cache, write a value with bit 11 set to the CACR. Bit 11 is the Clear Data Cache flag. This bit always reads as zero. Setting it clears the valid bits for all data cache entries. The most common use of this feature is for invalidating the cache when switching tasks in a multitasking operating system.

A single entry in the data cache may be invalidated with the Clear Data Cache Entry flag (bit 11). This bit always reads as zero. Writing a value to the CACR with bit 11 set to one causes a single longword in cache (specified by the Cache Address Register) to be invalidated. The Cache Address Register is the same as the corresponding register on the 68020 (described previously).

The Freeze Data Cache bit (9) disables replacement of cache entries. A cache hit caused by a write cycle will still alter the data in cache, however. A one value written to this bit freezes cache, and a zero value allows normal operation.

The Enable Data Cache bit (8) enables the data cache. This bit is set to zero when the 68030 is reset, disabling the data cache. Writing a value to the CACR with bit 8 set to one starts data cache operation. You can disable the data cache by clearing bit 8 in the CACR. Disabling the data cache does not invalidate the entries. If you disable the cache and subsequently re-enable it (without invalidating the entries), the old entries will be used. This is not normally desirable.

Bit 4 of the CACR is the Instruction Burst Enable flag. This bit is zeroed when the processor is reset. Setting this bit causes burst mode filling of the instruction cache. This bit is analogous to bit 12, which enables burst mode for the data cache.

The Clear Instruction Cache bit (bit 3) invalidates all entries in the instruction cache. This function is required in multitasking systems so that cache entries generated by one task are not erroneously used by another task. Software that switches from one task to another sets this bit in the CACR to avoid this problem. Like the freeze cache bits, there is no need to clear this bit, and it always reads back as zero.

A single cache entry may be invalidated using the Clear Instruction Cache Entry bit (bit 2) of the CACR. When this bit is set, the 68030 invalidates an entry in the instruction cache selected by the CAche Address Register (CAAR). This register is shown in Figure 2-29.

CACR bit 1 is the Freeze Instruction Cache bit. Setting this bit to one disables filling instruction cache entries. The cache continues to operate, but no entries are replaced. This bit is normally used by an in-circuit emulator to preserve cache contents during execution of emulator support software. The freeze bit always reads back as zero, and the cache is frozen each time a value with bit 1 set is written to the CACR. There is no need to clear the bit before freezing cache.

Bit 0 of this register is the Enable Instruction Cache bit. Setting this bit to zero forces continuous cache misses and suppresses filling cache entries. The cache is initially disabled when the processor is reset and must be explicitly enabled by software.

Exception Processing

Normal program execution on a 68000 family processor can be disturbed by a condition called an *exception.* Exceptions can be caused by devices outside the processor, or they may originate inside the processor (or a coprocessor). There are 255 possible types of exceptions on a 68000. Associated with each possible exception is an address, known as a *vector.* With the exception of the first vector, which consists of 2 long words, each vector consists of 1 longword. The vectors are arranged contiguously in memory, starting at a base address contained in the Vector Base Register (VBR) on the 68010/20/30, or at location zero on the 68000. Vectors are numbered from 0–255, with the address of the vector equal to the vector number times four (plus the VBR if not on a 68000). Note that vector number 0 is meaningless (see the discussion on RESET later in this chapter).

An exception caused internally by the processor, a coprocessor, or an illegal memory access is the result of a programming error. Such exceptions, or *faults,* happen when the error occurs and cannot be ignored. An exception caused by an external device is called an *interrupt.* Each interrupt has an associated 3-bit quantity called the *interrupt priority.* The interrupt mask field of the status register allows interrupts to be postponed. The processor will recognize only those interrupts whose priority exceeds the interrupt mask field in the status register, with the exception of level 7. A level 7 interrupt cannot be delayed. It is processed immediately, regardless of the status register contents. This interrupt is also called a *non-maskable* interrupt.

Exception Stack Frames

The 68000 CPUs push the program counter and status register on the relevant stack during an exception. Other information may also be present, as required by the exception in question. The stack frames have a number of different formats and are not compatible across the entire 68000 family. Figure 2-34 gives the default exception stack frame for the 68000 and the other members of this CPU family.

On a 68000, only the status register and program counter are saved. The remaining CPU chips save an additional word, which enables the exception handler to determine the exception vector address. This is useful in servicing multiple exceptions with a single piece of code. The word at offset 6 in the stack frame has the vector offset in the low 12 bits. This offset is the number of bytes from the beginning of the vector area (address zero on the 68000, and the contents of the Vector Base Register on the other CPUs). The top 4 bits of the word at offset 6 contain a *format code.* This field identifies the exception type. Most exceptions and all interrupts use format code 0, as pictured in Figure 2-34. There are also several other formats, which we will describe with the relevant exceptions.

The ReTurn from Exception (RTE) instruction is used to continue processing following an exception. Exception processing is, unfortunately, different for the different processors.

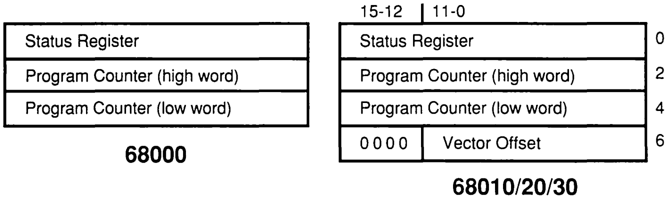

Status Register
Program Counter (high word)
Program Counter (low word)
68000
15-12
11-0
Status Register
0
Program Counter (high word)
2
Program Counter (low word)
4
0000
Vector Offset
6
68010/20/30

Figure 2-34. Default Exception Stack Frames

Faults

The 68000 family has a number of exceptions that may be classified as faults. These are normally the result of a programming error. These faults are as follows:

	Name	Vector	Address	CPUs
	Description			
BUSERR	2	$08	all	Accessing non-existent memory
Address error	3	$0C	all	Accessing memory on an illegal boundary
Illegal Instruction	4	$10	all	Attempt to execute an invalid opcode
Division by zero	5	$14	all	Attempt to divide something by zero
CHK / CHK2	6	$18	all	Range failure on a **CHK** or **CHK2** instruction
TRAPV	7	$1C	all	**TRAPV** (**TRAP**cc or **FTRAP**cc) instruction
Privilege Violation	8	$20	all	Executing a privileged instruction in user mode
Trace Trap	9	$24	all	Executing any instruction in trace mode
Line A Emulator	10	$28	all	Executing a word of the form $Axxx
Line F Emulator	11	$2C	all	Executing a word of the form $Fxxx
Coprocessor Protocol	13	$34	20/30	Hardware error in a coprocessor chip
Format Error	14	$38	10/20/30	Stack frame error
TRAP 0 - 15	32-47	$80-$BC	all	**TRAP** #n instruction
Unordered Condition	48	$C0	20/30	(68881/2 error, described in Chapter 4)
Inexact result	49	$C4	20/30	(68881/2 error, described in Chapter 4)
Division by Zero	50	$C8	20/30	(68881/2 error, described in Chapter 4)
Underflow	51	$CC	20/30	(68881/2 error, described in Chapter 4)
Operand Error	52	$D0	20/30	(68881/2 error, described in Chapter 4)
Overflow	53	$D4	20/30	(68881/2 error, described in Chapter 4)
Not a Number (NAN)	54	$D8	20/30	(68881/2 error, described in Chapter 4)
PMMU Configuration	56	$DC	20/30	(68851 error, described in Chapter 5)
PMMU Illegal Operation	57	$E0	20/30	(68851 error, described in Chapter 5)
PMMU Access Level	58	$E8	20/30	(68851 error, described in Chapter 5)

Each exception indicates a different type of error. The vectors not appearing in this table are described in the Interrupts section.

BUSERR The 68000 family of CPUs have an *asynchronous bus*. The processor outputs an address (and data for a write operation) and waits for a response. On the 68000 and 68010, the DaTa ACKnowledge (DTACK) signal (pronounced "dee-tack") lets the processor know that the transfer is complete. (The 68020 and 68030 have two such signals, called DSACK0 and DSACK1. These signals allow dynamic bus sizing.) If for some reason the transaction cannot be completed, asserting the Bus ERRor (BERR) instead of DTACK or DSACK0/1 causes the processor to abandon the access attempt. When this signal is asserted, the processor recognizes a BUSERR exception. This condition is known as "missing the bus" among UNIX aficionados.

This facility has two normal uses. Program errors can cause references to memory that does not exist. In most cases, the operating system terminates the offending program. The other usage is much more subtle. A system that supports virtual memory plays a shell game with programs, attempting to guess which areas of memory are needed when. Parts of the program that are not needed are left on disk. The result is that programs use less memory than they would otherwise. When the system guesses incorrectly, a device called a Paged Memory Management Unit (PMMU) causes a BUSERR exception. This situation is called a *page fault*. The virtual memory system loads the missing piece of memory and continues the program. This process is described in detail in Chapter 5.

In a virtual memory system, the CPU must be able to complete successfully an instruction that causes a page fault. The 68000 lacks this capability. The 68010, 68020, and 68030 have the ability to continue an instruction that causes a BUSERR exception. You can either re-try the failing access by continuing the instruction or supply the required data in the software which fields the BUSERR exception.

Addressing Error Accessing memory on an illegal boundary causes an exception through vector 3. On the 68000 and 68010, you get an addressing error by attempting a word or long access using an odd address. The 68020 and 68030 give addressing errors only when you try to execute an instruction on an odd boundary. In most cases, an addressing error is an indication of a serious software problem.

BUSERR/Addressing Error Stack Frame The stack frame for these two exceptions has the dubious distinction of being the least compatible area of the 68000 family. There are four separate stack formats: one for the 68000, one for the 68010, and two used by both the 68020 and 68030.

The 68000 and 68010 have a 16-bit data bus. The BUSERR/Addressing error stack frame for these two CPUs appears in Figure 2-35.

Offset 0 of the 68000 stack frame contains the function code bits (FC2–FC0) associated with the failing access. Bit 3 of this word is zero if the CPU was processing an instruction, and one if the error was detected by an external device. Bit 4 of this word is the Read/Write indicator. Zero indicates a write, and one indicates a

68000

15 … 5	4	3	2-0
▨	R/W	I/N	Access
Fault address (high word)			
Fault address (low word)			
First word of failing instruction			
Status Register			
Program Counter (high word)			
Program Counter (low word)			

(14 Bytes)

68010

15-12	11-0	
Status Register		0
Program Counter (high word)		2
Program Counter (low word)		4
1 0 0 0	Vector Offset	6
Special Status Word		8
Fault address (high word)		10
Fault address (low word)		12
Reserved		14
Data Output Buffer		16
Reserved		18
Data Input Buffer		20
Reserved		22
Instruction Input Buffer		24
Internal Information (16 Words)		26

(58 Bytes)

Figure 2-35. 68000/010 BUSERR/Addressing Error Stack Frames

read. Offsets 2 and 4 of the 68000 stack frame contain the address that caused the fault. Offset 6 has the first word of the failing instruction. The rest of the stack frame uses the normal 68000 exception format. The 68000 cannot re-try a failing access.

The first 8 bytes of the 68010 stack frame fit the standard exception format for this chip. The format code is 8 (1000 binary). The rest of the information in the stack frame is exception-specific. Offset 8 contains the Special Status Word (described below). Offsets 10–12 contain the address causing the exception. Offset 16 is the data to be output if the bus cycle is a write cycle. Offset 20 can be used to simulate a read completion using software. Offset 24 is the first word of the failing instruction. The remaining locations are not documented by Motorola.

The 68010 Special Status Word (offset 8 of the stack frame) contains information required to recover from a BUSERR exception. This register is shown in Figure 2-36.

The 68010 Special Status Word corresponds to offset 0 of the 68000 stack frame. Bits 2–0 contain the access code for the failed bus access. Bit 8 (RW) is zero if the access was a write, or one if the access was a read. Bit 9 (BY) is a one if the access was a byte access, and zero for a word access. Note that since the 68010 is a 16-bit processor, a long access is broken up into two word accesses. Bit 10 (HB) has meaning if bit 9 is set. On a byte access, bit 10 will be a one if the high byte (bits 15–8) are referenced, and a

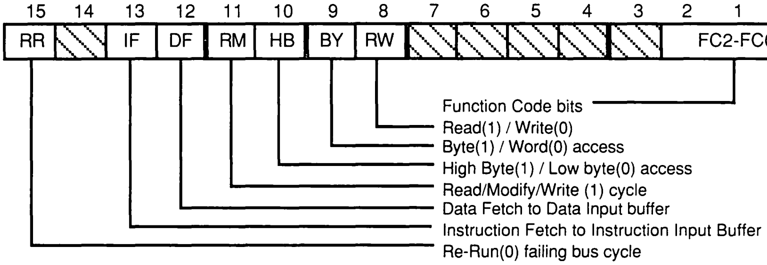

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
RR
IF
DF
RM
HB
BY
RW
FC2-FC0
Function Code bits
Read(1) / Write(0)
Byte(1) / Word(0) access
High Byte(1) / Low byte(0) access
Read/Modify/Write (1) cycle
Data Fetch to Data Input buffer
Instruction Fetch to Instruction Input Buffer
Re-Run(0) failing bus cycle

Figure 2-36. 68010 Special Status Register

zero for a reference to the low byte. Bit 11 (RM) is a one if the failing bus cycle is a Read/Modify/Write access. Bits 13–12 indicate the destination of a bus read operation. Bit 12 (DF) is set if the attempted access was a data read, and Bit 13 (IF) is set if the attempted access was an instruction fetch.

Bit 15 (RR) dictates processor action if the stack frame is used for an RTE instruction. If this bit is clear, the processor attempts to re-run the failed access. For systems that support virtual memory, the normal sequence of events is to fetch the missing memory, the address for which is in the fault address (offsets 10 and 12) of the stack frame, and issue an RTE to continue the faulting instruction. If the address is illegal or odd (for a word fetch), the access cannot be continued and the faulting program is usually terminated.

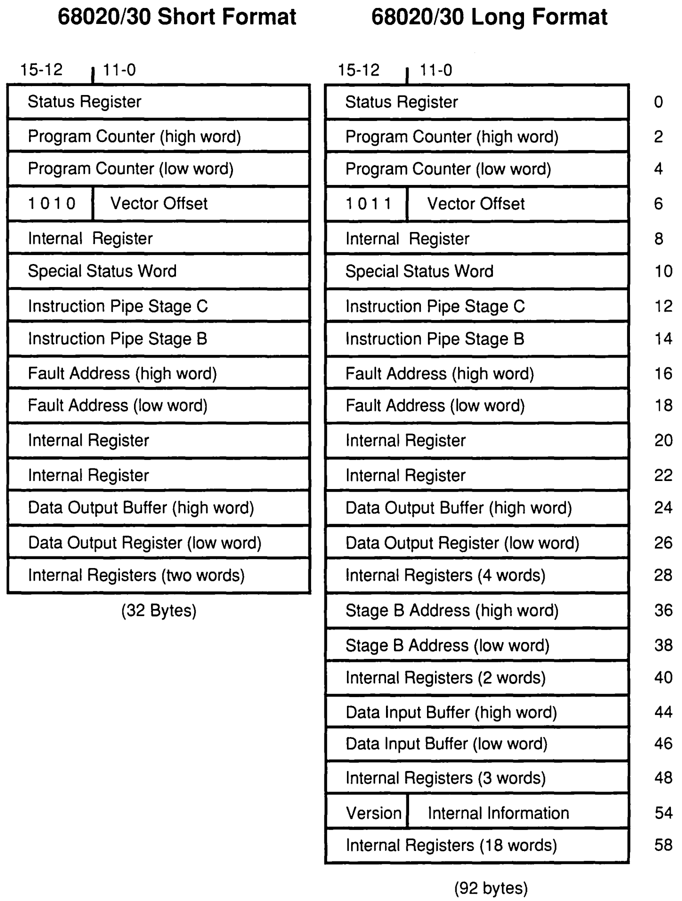
Another possibility is to perform the failed access in software. This is useful if you need to simulate the action of a missing piece of hardware. The sequence for simulating a bus cycle in software on the 68010 is as follows:

1. Determine if the cycle is a read or a write. The RW bit (8) indicates the cycle type. If this bit is set (a write), the data is in the Data Output Buffer offset (16) of the stack frame. Determine whether the cycle is a byte or word write (using bits 10–9). Do whatever is required for this type of write to the area you are simulating, then go to step 4.
2. For a read cycle, generate the data required at the faulting address (offsets 10–12). If the failing read was an instruction fetch, the data type will be word. Place the data in the Instruction Input Buffer (offset 24), then go to step 4.
3. For a data read cycle, use bits 10–9 to place the appropriate byte(s) in the Data Input Buffer (offset 20).

4. Set bit 15 of the Special Status Word and issue an RTE instruction.

Continuing a faulting program is normally done only for BUSERR exceptions. Addressing errors are usually an indication of a severe software problem. It is conceivable, however, that you could write a software handler for the 68010 that would emulate the ability of the 68020 and 68030 to access word and longword data on odd boundaries.

The 68020 and 68030 have a 32-bit bus. These chips use two stack frames for BUSERR and Addressing Error exceptions. These are called the short and long stack frames and are shown in Figure 2-37.



68020/30 Short Format (15-12 | 11-0)	68020/30 Long Format (15-12 | 11-0)	
Status Register	Status Register	0
Program Counter (high word)	Program Counter (high word)	2
Program Counter (low word)	Program Counter (low word)	4
1 0 1 0 | Vector Offset	1 0 1 1 | Vector Offset	6
Internal Register	Internal Register	8
Special Status Word	Special Status Word	10
Instruction Pipe Stage C	Instruction Pipe Stage C	12
Instruction Pipe Stage B	Instruction Pipe Stage B	14
Fault Address (high word)	Fault Address (high word)	16
Fault Address (low word)	Fault Address (low word)	18
Internal Register	Internal Register	20
Internal Register	Internal Register	22
Data Output Buffer (high word)	Data Output Buffer (high word)	24
Data Output Register (low word)	Data Output Register (low word)	26
Internal Registers (two words)	Internal Registers (4 words)	28
(32 Bytes)	Stage B Address (high word)	36
	Stage B Address (low word)	38
	Internal Registers (2 words)	40
	Data Input Buffer (high word)	44
	Data Input Buffer (low word)	46
	Internal Registers (3 words)	48
	Version | Internal Information	54
	Internal Registers (18 words)	58
	(92 bytes)	

Figure 2-37. 68020/68030 BUSERR / Addressing Error Stack Frames

The processor uses the short format when an exception occurs at an instruction boundary. It uses the long format for an exception that occurs during instruction execution.

The Special Status Word for the 68020 and 68030 appears in Figure 2-38.

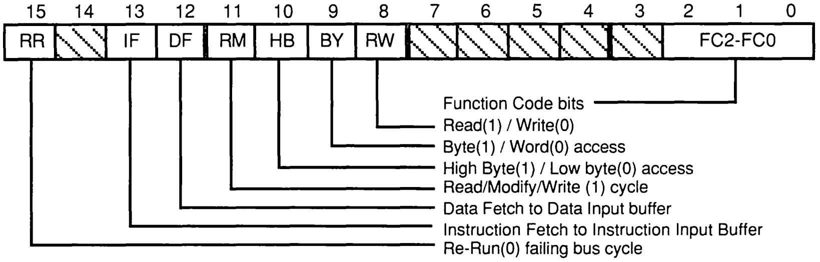

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
RR
IF
DF
RM
HB
BY
RW
FC2-FC0
Function Code bits
Read(1) / Write(0)
Byte(1) / Word(0) access
High Byte(1) / Low byte(0) access
Read/Modify/Write (1) cycle
Data Fetch to Data Input buffer
Instruction Fetch to Instruction Input Buffer
Re-Run(0) failing bus cycle

Figure 2-38. 68020/68030 Special Status Register

Note that this word has a different format than the corresponding word on the 68010. Some of the fields have identical functions, but occupy different positions. Bits 2–0 (FC2–FC0) contain the function code for the failing access. This field is identical to the one on the 68010. Bits 5–4 (SIZE) indicate the size of the bus transaction: 00 means byte, 11 means word, and 10 means long. Bit 6 (RW) indicates whether the attempted access was a read (1) or a write (0). This same bit appears in position 8 on the 68010. Bit 7 (RM), if a 1, indicates a read/modify/write data transaction. This bit corresponds to bit 11 on the 68010. Bit 8 (DF) indicates a data fault for the failing bus cycle. Note that this bit is different from the Data Fetch bit on the 68010. If the Data Fault bit is set, the bus cycle is the result of an attempt to read or write data. Leave this bit set to attempt re-running the cycle using an RTE instruction.

Bits 15–12 govern recovery from faults during instruction fetches. The 68020 and 68030 have a two-stage instruction pipeline. The stages are called "B" and "C". When a fault occurs during instruction fetch, the processor defers taking the exception until the instruction is actually required for execution. This prevents an erroneous exception caused by trying to pre-fetch the instruction after a branch contained in the last valid location in memory (the 68000 and 68010 will generate this erroneous exception). When the exception is taken, bits 14 (FC) and 15 (FB) indicate which pipeline stage lacks data. Bits 13 (RC) and 12 (RB) indicate which bus cycles to re-run using an RTE instruction.

As with the 68010, you can continue the faulting instruction by either re-running the failed bus cycle in hardware or by simulating the access in software. To re-run the failing bus cycle, simply issue an RTE instruction. The stack frame contains the appropriate values for this operation when the exception occurs. You must, of course, remedy the problem which caused the fault, or you will experience another exception.

To simulate the failing cycle in software, determine if the fault occurred during an instruction fetch, a data access, or both. For a data fetch, replace the Data Input Buffer (offsets 44–46) with the correct data. For word or byte accesses, the data should be in the low-order word or byte. For a data store, the Data Output Buffer (offsets 24–26) contains the data to be written at the fault address. The data in a write operation is also right-justified in the data output buffer. To resume execution of the faulting program, clear the DF bit in the Special Status word and execute an RTE instruction.

Read/modify/write cycles pose a slightly more difficult situation. You must simulate the operation of the TAS, CAS, or CAS2 instruction before performing the RTE. These instructions are the only ones that generate a read/modify/write bus cycle. If the DF bit in the Special Status Word is clear and the RM bit is set on an RTE, the processor expects the instruction to be complete.

Simulating a failed instruction fetch requires a similar procedure. The FB and FC bits indicate which pipeline stage caused the fault. Both the B and C stages or just the C stage may have caused the error. Simply replace the Instruction Pipe C (and B) offsets with the appropriate instruction stream values, then clear the RC (and RB) bits in the Special Status Word and RTE.

Some words of caution on simulation of failed accesses: First, you may change only the RB, RC, and DF bits in the Special Status Word. Also, if your system has an MMU, you must retrieve or store the data using the correct address space (as dictated by the S bit in the CPU status register).

Illegal Instruction Attempting to execute a bit pattern that is not a valid 68000 instruction results in an exception through vector 4. Illegal instructions include undefined opcodes, as well as invalid opcode / addressing mode combinations. The pattern $4AFC is guaranteed to produce an illegal instruction exception on all 68000 family members. Opcodes beginning with $A or $F in bits 15–12 are also illegal, but there are separate vectors for these opcodes, the A-line, and F-line traps (described below).

The illegal instruction exception uses the default stack frame (shown in Figure 2-34). The stacked value of the program counter points to the opcode which caused the exception.

This exception is quite often used by debugging software to gain control at an arbitrary point in a program being debugged. The debugger places an illegal instruction at the point where execution is to pause, and begins execution. The illegal instruction exception returns control to the debugger before the instruction is executed. To resume execution, you must replace the illegal instruction with the original contents before proceeding. Since the program counter points to the illegal instruction, no adjustment is required.

The 68020 and 68030 use the opcodes $4848–$484F as breakpoint instructions for use with external hardware, such as an In-Circuit Emulator. When the processor attempts to execute one of these instructions, it runs a bus cycle called the Breakpoint Acknowledge cycle. External hardware can supply an instruction to be executed instead of the breakpoint. If there is no response, the processor undertakes an Illegal Instruction exception.

Division by Zero All 68000 family members have instructions (DIVS and DIVU) that divide a 32-bit number by a 16-bit number. The 68020 and 68030 have two additional instructions for 32-bit division, DIVSL and DIVUL. An attempt to use zero as a divisor in any of these instructions results in a Division by Zero Exception, through vector number 5. On the 68000 and 68010 chips, this exception uses the default stack frame, as shown in Figure 2-34. The 68020 and 68030 use a 12-byte stack frame, as shown in Figure 2-39.

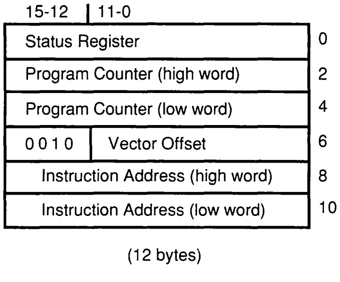

15-12
11-0
Status Register
0
Program Counter (high word)
2
Program Counter (low word)
4
0 0 1 0
Vector Offset
6
Instruction Address (high word)
8
Instruction Address (low word)
10
(12 bytes)

Figure 2-39. 68020/68030 Format 2 Exception Stack Frame

The format code for this exception (bits 15–12 of offset 6) is 2. This stack frame is similar to the default exception stack frame, with the addition of the address of the faulting instruction at offsets 8–10. The stacked program counter value points to the next instruction to be executed for all exceptions using this stack frame.

CHK/CHK2 Exception The 68000 family has an instruction (CHK) used for range checking at runtime. CHK is limited to testing values against zero and an upper bound. The 68020 and 68030 have an additional instruction, CHK2, that can test a general range of values. The purpose of these instructions is to cause an exception if the value tested is out of range. CHK instructions are normally used by high-level languages to prevent array subscript errors from destroying other data.

An out of range condition causes an exception through vector 6. On the 68020 and 68030, this exception uses a format 2 stack frame, as shown in Figure 2-39. The default stack frame (Figure 2-34) is used on the 68000 and 68010. The stacked PC value points to the instruction following the CHK instruction.

TRAPV/TRAPcc Exception A related exception is the conditional TRAP exception, which uses vector 7. All 680x0 processors have a TRAPV instruction that causes an exception if the V bit of the status register is set. The 68020 and 68030 have an additional instruction that allows a conditional trap on any combination of the condition codes in the status register, similar to a conditional branch. These instructions are also used for error detection at runtime, and are more general than the CHK instructions.

On the 68020 and 68030, these traps use a format 2 stack frame, as shown in Figure 2-39. The 68000 and 68010 use the default stack frame, as shown in Figure 2-34. The stacked value of the program counterpoints to the word following the first word of the instruction.

Privilege Violation Exception Attempted execution of a privileged instruction in user mode results in an exception through vector 8. This exception uses the default exception stack frame (Figure 2-34) on all processors. The stacked value of the program counter is the address of the instruction that caused the exception.

Trace Trap Setting status register bit 15 on all 680x0 CPUs causes an exception following completion of the next instruction. On the 68020 and 68030, status register bit 14 can be used to trap on the next instruction, such as a branch or RTE instruction that causes a break from sequential execution.

This feature is normally used for a debugger, allowing a programmer to step through a faulty program one instruction at a time. The trace exception uses vector 9. On the 68020 and 68030, this exception uses a format 2 stack frame, as shown in Figure 2-39. The default stack frame (Figure 2-34) is used on the 68000 and 68010. The stacked value of the program counterpoints to the next instruction to be executed.

Line A Emulator Trap Executing a word with $A in bits 15–12 causes an exception through vector 10. This feature allows extending the instruction set of the processor, using values in the remaining 12 bits of the opcode to indicate a function to be performed by the exception handler. The Macintosh uses this particular trap to allow application programs to call system routines, for instance.

The advantage of this technique is that it allows up to 4096 different functions to be specified using a single 16-bit word. The application also need not know where the system software resides in memory. The disadvantage is that more time is required to execute a given function, due to overhead software required to call the proper subroutine based on the contents of the A-line instruction.

The Line A trap uses a default stack frame (shown in Figure 2-34). The value of the stacked program counterpoints to the opcode causing the trap. This value must be incremented by two before continuing the faulting program.

Line F Emulator Trap A similar function occurs for opcodes with $F in bits 15–12. The 68020 and 68030 use the F-line opcodes for coprocessor extensions to the basic CPU instruction set. The 68881/68882 (described in Chapter 4) and the 68851 (described in Chapter 5) both use this technique. If an F-line opcode is not valid for any coprocessor in a system, the processor takes an F-line trap through vector 11.

F-line traps have the same size advantages and speed disadvantages as the A-line trap. There is, however, another benefit. By writing software that emulates the action of a coprocessor, it is possible to make the same program run both on a system with a coprocessor and on a system without the coprocessor. The only difference is speed. A hardware vendor can offer the coprocessor as an extra cost option that improves performance, without invalidating the customer's existing software investment. The software emulation can also allow a program using F-line instructions to run on a 68000 or 68010 machine as well. This allows the same software to run on machines within the same product line.

The Line F trap uses a default stack format (shown in Figure 2-34). The value of the stacked program counter points to the F-line opcode which caused the trap. This stacked value must be incremented by two before continuing the faulting program.

Coprocessor Protocol Another exception possible on an F-line instruction is the coprocessor protocol exception through vector 13. This exception results when the 68020 or 68030 detects an erroneous response from a coprocessor. A protocol exception normally indicates some kind of hardware problem or an erroneous attempt to access the coprocessor's interface registers directly with a MOVES instruction.

Coprocessor protocol exceptions use a format 9 stack frame. This stack frame is 20 bytes in length and appears in Figure 2-40.

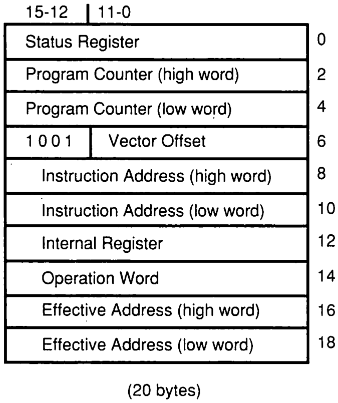

15-12
11-0
Status Register
0
Program Counter (high word)
2
Program Counter (low word)
4
1 0 0 1
Vector Offset
6
Instruction Address (high word)
8
Instruction Address (low word)
10
Internal Register
12
Operation Word
14
Effective Address (high word)
16
Effective Address (low word)
18
(20 bytes)

Figure 2-40. Format 9 Exception Stack Frame

The stacked program counterpoints somewhere in the middle of the instruction, depending on where the error was detected. The instruction address points to the beginning of the instruction, and the first word of the instruction is contained in the operand word field. The effective address field gives the memory address of the instruction's first effective address operand (if any).

Format Error Exception (Not on 68000.) An RTE on the 68010, 68020, or 68030 with an unknown stack frame format code (bits 15–12 of offset 6) causes an exception through vector 14. Format codes 0 and 8 are valid on the 68010. The 68020 and 68030 utilize format codes 0, 1, 2, 9, 10, and 11. All others cause a format error exception, which uses a format 0 exception stack frame, shown in Figure 2-34. The stacked value of the PC points to the RTE or cpRESTORE instruction that caused the fault.

TRAP Instruction Exception The TRAP #n instruction causes a fault through vectors 32–47. There are 16 trap instructions (0–15) and 16 corresponding vectors. These instructions are normally used as a way for an application to request service from the operating system. The application normally runs in user mode, with the operating system in supervisor mode. The TRAP instruction provides a clean way for the user mode program to invoke supervisor mode software at an entry point controlled by the supervisor mode software. A subsequent RTE will return the processor to user mode. This technique provides a secure way for a nonprivileged program to execute privileged software without compromising the integrity of the system.

TRAP exceptions all use the default stack formats shown in Figure 2-34.

Coprocessor-Detected Faults The 68881/2 floating point unit, the 68851 memory management unit, and other coprocessors can also cause a fault. The 68881/2 can cause an exception using vectors 48–54, and the 68851 can cause an exception using vectors 56–58. The 68000 and 68010 do not support coprocessors and do not have these exceptions. The exceptions through vectors 57 and 58 apply only to the 68020 module feature. The 68030 does not have this feature and does not have the exceptions. Coprocessor-related exceptions are fully described in chapters 4 and 5.

Interrupts

68000-family chips have three signals that allow external devices to request an interrupt. These are the IPL2, IPL1, and IPL0 pins. These three signals yield eight combinations. The signals can be either asserted (low logic level) or not (high logic level). The combination where none of the IPL pins are asserted means that no interrupts are being requested. Any other combination means that some device has requested an interrupt at a priority equal to the state of the IPL pins. If the interrupt mask in the Status Register is less than the requested interrupt priority, the interrupt will be taken. If not, the interrupt will be postponed until the Status Register is changed. Most software attempts to minimize the time interrupts are disabled. This produces the shortest response time to external events.

To recognize an interrupt, the CPU runs a bus cycle known as an *interrupt acknowledge* cycle. The processor outputs the interrupt level requested on bits A3–A1 and waits for a reply. Three replies are possible. The output device can supply a vector, can request autovectoring, or there can be no response. Lack of response is indicated by the BUSERR signal, resulting in a *spurious interrupt* exception through vector 24.

A vectored interrupt is the result of the device supplying a vector number on the data bus. Autovectored interrupts allow a device to specify use of a default vector. There is such a vector per priority level (1–7). Use of autovectoring reduces external hardware costs. This is the most popular type of interrupt mechanism for personal computers.

One abuse of the autovectoring technique deserves special mention. Since there are three interrupt request lines, systems with only three interrupting devices can attach a device to each request line. In the normal case, there are three interrupts: 1, 2, and 4. However, if two devices request an interrupt simultaneously, you can also have interrupts with priorities 3 (1 and 2), 5 (1 and 4), 6 (2 and 4) and 7 (1, 2, and 4). In most cases, simply duplicating the vectors to allow the different interrupt combinations is sufficient. However, level 7 is non-maskable, and interrupts can never be completely disabled on such a system. Early versions of the Macintosh computer used this technique.

Interrupt Vectors The vectors used by interrupts are as follows:

Name	Vector	Address	CPUs	Description
Reset Initial SP	0	$00	all	Initial SP for external Reset
Reset Initial PC	1	$04	all	Initial PC for external Reset
(Exceptions)	2-11	$08-$2C	all	See previous section on Faults
(Unassigned, Reserved)	12	$30	all	Currently Unused
(Exceptions)	13-14	$34-$38	all	See previous section on Faults
Uninitialized Interrupt	15	$3C	all	Uninitialized Interrupt vector
(Unassigned, Reserved)	16-23	$40-5C	all	Currently Unused
Spurious Interrupt	24	$60	all	Interrupt protocol error
Level 1 Autovector	25	$64	all	Autovector interrupt at priority 1
Level 2 Autovector	26	$68	all	Autovector interrupt at priority 2
Level 3 Autovector	27	$6C	all	Autovector interrupt at priority 3
Level 4 Autovector	28	$70	all	Autovector interrupt at priority 4
Level 5 Autovector	29	$74	all	Autovector interrupt at priority 5
Level 6 Autovector	30	$78	all	Autovector interrupt at priority 6
Level 7 Autovector	31	$7C	all	Autovector interrupt at priority 7
(Exceptions)	32-58	$80-$E8	all	See previous section on Faults
(Unassigned, Reserved)	59-63	$EC-FC	all	Currently Unused
User Interrupt Vectors	64-255	$100-$3FC	all	Reserved for machine-specific interrupts

Reset All 68000 family chips have an external line called RESET. When this line is asserted, the processor stops execution and performs the following actions:

1. The Trace and Master bits in the status register are cleared. The Supervisor bit and Interrupt Mask bits are set. SR takes on the value $27xx (supervisor mode, interrupt stack, and interrupts disabled).
2. The Vector Base Register is set to zero.
3. On the 68020 and 68030, all caches are disabled and all entries are marked invalid. On the 60830, the Burst Mode and Write Allocate features are also disabled.
4. The Interrupt Stack Pointer (ISP) is loaded from location $0, and the Program Counter from location $4. Since the Vector Base Register is cleared by RESET, locations 0 and 4 must exist when RESET is asserted.

No information is pushed onto the stack, and the original contents of the Interrupt Stack Pointer and Program Counter are lost. On the 68000 and 68010, this stack pointer is called the Supervisor Stack Pointer (SSP).

For a 68020 system with a 68851 MMU or a 68030, the MMU Address Translation Cache is not affected by RESET.

Uninitialized Interrupt Vector Vector number 15 (address $3C) is reserved for devices that can be programmed to interrupt through any vector. The device should use vector 15 if the CPU has not specified which vector to use. (This is normally a programming error.) Note that this standard applies to vectored interrupts only and cannot be used with autovectored interrupts.

Spurious Interrupt Vector 24 (address $60) is used when a vectored interrupt is requested but no vector is supplied. This condition is brought about by the assertion of a BUSERR indication when the processor requests the vector from the peripheral. (This is usually an indication of a hardware problem.)

Autovectored Interrupts As mentioned previously, a device can request an autovectored interrupt during the interrupt acknowledge cycle. The device asserts the AVEC signal instead of placing a vector on the bus. There is one autovector for each priority level (1–7). These priority levels correspond to vectors 25–31 (addresses $64–$7C). This technique is widely used in less expensive computers because it costs less to implement.

Interrupt Stack Frames Interrupts on the 68000 and 68010 chips always use the default exception stack frames shown in Figure 2-34. The 68020 and 68030 use format 0 (Figure 2-34) or format 9 (Figure 2-40) stack frames on the Interrupt Stack if the processor is interrupted from user mode or if the processor is using the Interrupt Stack (bit 12 of the Status Register is zero). The processor uses format 0 unless the interrupt occurs in the middle of a coprocessor instruction. An interrupted coprocessor instruction uses format 9. When the processor is in supervisor mode and using the Master Stack (bits 13 and 12 of the Status Register are both set), two stack frames are generated.

When the processor is interrupted while on the Master Stack, the normal interrupt stack frame is built on the Master Stack. An additional stack frame is built on the Interrupt Stack. This is a format 1 stack frame, and is shown in Figure 2-41.

This second stack frame is also called a *throwaway* stack frame. The Status Register is reloaded from this stack frame, the rest of the frame is discarded, and a second stack frame (usually located on the Master Stack) is used to complete the RTE. The other fields in the throwaway (ISP) frame are copies of those on the master stack. Note that the use of the Status Register image from the throwaway frame allows use of the stack frame on the Master Stack (MSP) instead of the Interrupt Stack (ISP). This is the normal case. You can also use this type of stack frame to reference a second stack frame on the interrupt stack (ISP) by setting the status register M bit to zero in the throwaway frame.

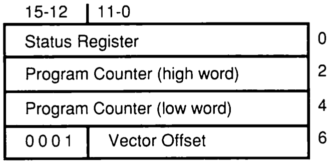


Figure 2-41. Format 1 (Throwaway) Exception Stack Frame

Coprocessor Operations

The 68020 and 68030 allow extensions to the instruction set to be implemented in a separate chip called a *coprocessor*. There are currently three such chips supplied by Motorola: the 68881/68882 floating point chips, and the 68851 Memory Management Unit (MMU). On the 68030, a subset of the 68851 is built-in.

Coprocessor Instruction Format

Coprocessors share a common instruction format, as shown in Figure 2-42.

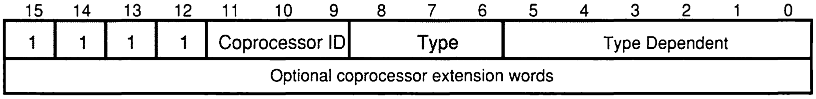


Figure 2-42. Coprocessor Instruction Format

Note that the top 4 bits all have a value of 1. Coprocessor instructions executed on a machine without a coprocessor will cause a Line F Emulator trap through vector 11. This architecture allows you to write a single program that works with or without a coprocessor, using software emulation for machines without the coprocessor. This technique is widely used for 68881/68882 floating point software, for instance.

The coprocessor ID field (bits 11–9) in the first word of the instruction determines which coprocessor will service the instruction. Motorola has reserved the first 6 bit combinations (000–101) for Motorola-supplied coprocessors. The 68851 chip has been assigned ID 000, and the 68881/68882 chips have ID 001. Bit combinations 110–111 may be used by hardware manufacturers other than Motorola.

The instruction type field (bits 8–6) indicates the operation to be performed by the coprocessor. This field, in conjunction with bits 5–0, specifies the coprocessor instruction. There are eight such instructions:

Type	**Type Dependent**	**Instruction**	**Notes**
000	Effective Address	cpGEN	General purpose instruction set extension
001	001 + Data Register	cpDBcc	Similar to DBcc looping instruction
001	Effective Address	cpScc	Similar to Scc conditional instruction
001	111 + Length code	cpTRAPcc	Similar to TRAPcc instruction
010	Condition Selector	cpBcc.W	16-bit offset conditional branch
011	Condition Selector	cpBcc.L	32-bit offset conditional branch
100	Effective Address	cpSAVE	Save coprocessor internal state
101	Effective Address	cpRESTORE	Restore saved internal state

These instructions are described in detail as part of Chapter 3. Note that there are three instructions with a type code of 001. The upper 3 bits of the type dependent field are used to make the distinction. This effectively disallows the use of the Address Register Direct, Absolute, PC-relative, and Immediate Addressing modes with the cpScc instruction. Since the Scc operand is a destination, the PC-relative and Immediate modes are unusable anyway.

The cpGEN instruction format is the most flexible. cpGEN allows the addition of new data types and more complicated instructions than those provided by the 680x0. cpGEN is the only coprocessor instruction type that allows the coprocessor and the main processor to execute independently. The other formats are more specialized.

The cpDBcc, cpScc, cpTRAPcc, and cpBcc instructions provide coprocessor conditional instruction capabilities compatible with those provided by the main processor. A 68000 condition is a 4-bit field, as explained in Chapter 3. Coprocessor conditions have a 6-bit field that is interpreted only by the coprocessor.

The cpSAVE and cpRESTORE instructions permit saving a coprocessor's internal state and later restoring it. This is useful in multitasking operating systems like UNIX, where more than one application may be using a coprocessor. The operating system must save the current state and restore a previous state when switching from one application to another. This save/restore feature is widely used for the 68881/68882 floating point chips.

The coprocessor extension words are interpreted only by the coprocessor. These are defined for each coprocessor used in a given system.

Coprocessor-Generated Exceptions

Coprocessors can generate exceptions, before, during, and after the execution of a coprocessor instruction. The coprocessor specifies the exception vector and the type of exception.

The pre-instruction exception serves two primary purposes: If a coprocessor detects illegal values in one of the instruction fields, it will issue a pre-instruction exception. This usage corresponds to the 68000 illegal instruction exception. The second use concerns the cpGEN instruction. When a coprocessor allows the main processor to execute concurrently, errors in the coprocessor are deferred until the next coprocessor instruction. The pre-instruction exceptions use a format 0 stack frame, shown in Figure 2-33. The stacked value of the PC points to the beginning of the coprocessor instruction so that an RTE attempts to run the instruction again.

The mid-instruction exception uses a format 9 exception stack frame, as shown in Figure 2-40. If the main processor does not modify the stack frame, an RTE instruction will attempt to continue the coprocessor operation.

A post-instruction exception uses a format 2 exception stack frame, as shown in Figure 2-39. This type of exception is used when the coprocessor has aborted or finished the instruction in question. An RTE will cause execution to resume at the next instruction.

CHAPTER

3

Instruction Set

"I hate this machine. I wish that they would sell it.
It doesn't do what I want, but only what I tell it."
— Programmer's Lament

Introduction

This chapter presents details of the 68000 family CPU instruction set, including the general-purpose coprocessor instructions. The floating point and MMU-specific coprocessor instructions are presented in Chapters 4 and 5.

Instruction Description Format

Each instruction description contains a header, a description, a list of valid addressing modes, the instruction format, and an example of instruction execution. Each item is described below. The addressing modes are omitted for certain instructions, and some instructions may have additional material.

Header

The header consists of the instruction name, assembler syntax, and an expansion of the instruction mnemonic. For example, consider the header for the ABCD instruction:

ABCD Instruction

ADD Binary Coded Decimal with extend

Syntax: ABCD Dy,Dx
ABCD -(Ay),-(Ax)

The instruction mnemonic is at the top left corner of the header, with the mnemonic expansion directly below. The right-hand side of the header gives the valid assembly language forms of the instruction. The ABCD instruction has two such forms — one that adds a data register to a data register (Dy,Dx), and one that uses two address registers (-(Ay),-(Ax)). Symbols used in the syntax descriptions are as follows:

An	A single address register
Ax,Ay	Used when two address registers are required in a description
#<data>	Immediate data
Dn	A single data register
Dx,Dy	Used when two data registers are required in a description
<ea>	An effective address operand
<label>	A label on an instruction, used as a branch target address
Rn	A single address or data register
Rx,Ry	Used when two registers (address or data) are required in a description

Description

The instruction description consists of a brief overview of the instruction operation. The following diagram is also included:

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	0	U	-		•	•	•		•	•	•	•

The condition codes section gives the effect of the instruction execution on each of these status bits. There are six possible indicators:

-	The bit is not affected by the instruction
*	The bit is conditionally set by the instruction
0	The bit is always cleared by the instruction

1 The bit is always set by the instruction
U The bit is undefined following instruction execution
? The bit may (or may not) be affected by a coprocessor

The text immediately following the diagram describes how the condition codes are affected. The other two sections of the diagram show which data sizes the instruction supports and which CPUs support the instruction.

The Sizes field describes which data sizes the instruction supports. You select the appropriate size using the .B, .W, and .L suffixes for the mnemonic. For instance, a 32-bit ADD instruction uses the instruction mnemonic ADD.L. Branch instructions (Bcc and BSR) use a .S (short) suffix rather than .B. (Some assemblers will accept either suffix, however.)

The CPUs field indicates which members of the family implement the instruction. As a rule, instructions are upwardly compatible. If an instruction is on the 68000, it remains on all the others. The exceptions to the rule are the CALLM and RETM instructions, which are only implemented in the 68020.

Addressing Modes

A second diagram indicates which addressing modes are valid with the instruction. This diagram has columns for both source and destination operands. The table from the ADD instruction is shown below:

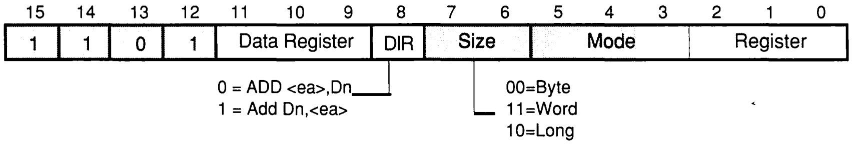
Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•		xxx.W	111	000	•	•
An *	001	An	•		xxx.L	111	001	•	•
(An)	010	An	•	•	#<data>	111	100	•	
(An)+	011	An	•	•					
-(An)	100	An	•	•					
(d16,An)	101	An	•	•	(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•	•	(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•	•	(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•	•	([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•	•	([bd,PC],Xn,od)	111	011	•	

* Word or long only ☐ 68020/030 only

Both source and destination addressing modes are indicated, even if the instruction does not support two full-function addressing modes. Most two-operand instructions support one generalized addressing mode and one specialized addressing mode.

Instruction Format

The third item in an instruction description is a diagram of the binary form of the instruction. The diagram for the ADD instruction is shown below.



Instructions are shown as one or more 16-bit quantities. The shadings indicate bit groupings for hexadecimal. The 68000 architecture does not allow quick translation between an instruction and the hexadecimal equivalent. Instruction fields are explained in text following the format diagram. Because they are present in most instructions in bits 5–0, the effective address fields (Mode and Register) are not described.

Example

To illustrate the use and binary layout of the instruction, the last part of the description is an example. In the example, we take a sample instruction and describe both its binary form and what happens upon execution. For instance, in describing the ADD instruction, we use ADD.W D0,D1 for the example. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Data Register			DIR	Size		Mode			Register		
1	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0

Note that the instruction format is repeated to better illustrate how the fields are constructed for the sample instruction.

The output of a symbolic debugger indicates what happens when the instruction executes. The ADD instruction example contains the following lines:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 0000ffff 00000001 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADD$      ADD.W  D0,D1
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2715 -- -- SI PL=7 X-Z-C
D0: 0000ffff 00000000 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADD$+002  NOP
```

The debugger output shows the contents of all the general-purpose registers, the status register, and the program counter. The debugger is symbolic. The label used on the example instruction is the instruction mnemonic with a dollar sign ($) appended. This label is ADD$ for the ADD instruction. The offset in the **After:** display shows the instruction length (in this case, 2 bytes).

All registers unused by the example will contain their own name, with register A4 containing hexadecimal 000000A4, for instance. The status register will normally have no condition code bits set before the sample instruction. There are exceptions to this rule, however, and some examples require one or more bits to be set.

In the **After:** display, anything changed by the instruction execution is underlined. In the example above, the X, Z, and C condition codes and the contents of register D1 were changed. The program counter always changes as a result of instruction execution. It is never underlined unless the instruction caused a branch. Only the Bcc, BSR, DBcc, JMP, JSR, RTS, RTR, and RTE instructions can cause such a branch.

The actual instruction sets for this chapter start on the next page.

ABCD Instruction

ADD Binary Coded Decimal with extend

Syntax: ABCD Dy,Dx
ABCD -(Ay),-(Ax)

Description

The ABCD instruction adds two 8-bit binary-coded decimal (BCD) numbers to produce an 8-bit result. The eXtend condition code bit participates in the addition and is set by the result. The instruction can have two forms:

1. Add a data register to another data register
2. Add two bytes in memory using the Address Register Predecrement addressing mode

The 68000 big-endian data format dictates the form of multiple-precision operations with the ABCD instruction. The memory-to-memory form must begin at the highest-addressed byte and work back to the lowest-addressed byte, using predecrement addressing.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	U	*	U	*		•				•	•	•	•

X	Set if decimal carry was generated. Cleared otherwise.
N	Undefined.
Z	Cleared if the result is non-zero. Unchanged otherwise.
V	Undefined.
C	Set if decimal carry was generated. Cleared otherwise.

You must ensure that the X bit (decimal carry) is in the appropriate state for this instruction. This bit must be cleared before a single-byte operation or the first in a series of multiple-byte operations. The second and subsequent operations in a multiple-byte operation use the X bit setting generated by the previous step.

The behavior of the Z condition code follows the 68000 standard for multiple-precision arithmetic. You normally set this bit before beginning a multiple-precision operation. If it is still set at the end of the operation, the entire result is zero.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Register Rx			1	0	0	0	0	R/M	Register Ry		

Bits 11-9 and 2-0 contain the two register numbers indicated in the syntax description. Bit 3 (R/M) indicates whether the instruction is a register-to-register operation (R/M = 0) or a memory-to-memory operation (R/M = 1).

Example This example adds the BCD quantity 11 contained in D1 to the BCD quantity 88 contained in register D0. The result is 99, replacing the contents of the destination register (specified by the Rx field). This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Register Rx			1	0	0	0	0	R/M	Register Ry		
1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1

The hex value is $C101. Executing the instruction yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 00000088 00000011 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ABCD$       ABCD       D1,D0
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 00000099 00000011 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ABCD$+002   NOP
```

ADD Instruction

ADD Binary

Syntax: ADD <ea>,Dn
ADD Dn,<ea>

Description

The ADD instruction performs a two's complement addition between a data register and an effective address operand.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•	•	•		•	•	•	•

X Set if a carry is generated. Cleared otherwise.
N Set if the result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a carry is generated. Cleared otherwise.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•		xxx.W	111	000	•	•
An *	001	An	•		xxx.L	111	001	•	•
(An)	010	An	•	•	#<data>	111	100	•	
(An)+	011	An	•	•					
-(An)	100	An	•	•					
(d16,An)	101	An	•	•	(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•	•	(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•	•	(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•	•	([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•	•	([bd,PC],Xn,od)	111	011	•	

* Word or long only ☐ 68020/030 only

When you specify an ADD instruction with an address register as a destination, many assemblers automatically generate the ADDA instruction. A similar situation exists with immediate data. You can specify an immediate mode source and a destination mode other than Dn, and the assembler will generate an ADDI instruction.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Data Register			DIR	Size		Mode			Register		

0 = ADD <ea>,Dn
1 = ADD Dn,<ea>

00=Byte
01=Word
10=Long

Example The ADD.W D0,D1 instruction adds the low 16 bits of D0 to the low 16 bits of D1. The upper 16 bits of D1 are unchanged. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Data Register			DIR	Size		Mode			Register		
1	1	0	1	0	0	1	0	0	1	0	0	0	0	0	0

The hex value of the instruction is $D240. Executing this instruction yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 0000ffff 00000001 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADD$        ADD.W      D0,D1
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2715 -- -- SI PL=7 X-Z-C
D0: 0000ffff 00000000 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADD$+002    NOP
```

Note the condition code settings. The X and C bits indicate a carry into the seventeenth bit of the result. This bit does not exist on a word operation. The Z bit indicates that the result is zero.

ADDA Instruction

ADD Address

Syntax: ADDA <ea>,An

Description

The ADDA instruction adds an effective address operand to an address register. Although the size may be specified as word or long, the entire address register is always affected. Both word and long operations produce a 32-bit result. A word operand is sign-extended to 32 bits and added to the entire address register.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-			•	•		•	•	•	•

The ADDA instruction does not change any condition code bits. You can perform address computations without losing the condition code settings of a previous data calculation.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•		xxx.W	111	000	•	
An	001	An	•	•	xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100	•	
(An)+	011	An	•						
-(An)	100	An	•						
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Register			Size	1	1	Mode			Register		

Bits 11-9 contain the destination address register number. Bit 8 (Size) is zero for word operations and one for long operations.

Example The ADDA.W D0,A0 instruction adds the low 16 bits of D0 to register A0. The contents of D0.W are sign-extended to 32 bits, and added to all 32 bits of register A0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Register			Size	1	1	Mode			Register		
1	1	0	1	0	0	0	0	1	1	0	0	0	0	0	0

The hex value of the instruction is $D0C0. Executing the instruction yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 0000ffff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 00000001 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADDA$       ADDA.W            D0,A0
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 0000ffff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 00000000 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADDA$+002   NOP
```

Note that the data in this example is the same data used with the ADD instruction example. There are no changes to the condition codes in this example, however.

ADDI Instruction

ADD Immediate

Syntax: ADDI #<data>,<ea>

Description

The ADDI instruction adds immediate data to an effective address operand.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•	•	•		•	•	•	•

X Set if a carry is generated. Cleared otherwise.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Set if an overflow was generated. Cleared otherwise.
C Set if a carry is generated. Cleared otherwise.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100	•	
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	Size		Mode			Register		
Word data (16 bits)								Byte data (8 bits)							
Long data (32 bits)															

The size field indicates the operation size: 00 for byte, 01 for word, and 10 for long. There are 2 or 4 bytes of immediate data that follow the instruction in memory. Byte and word operations have 16 bits (2 bytes) of immediate data. Long operations have 32 bits (4 bytes).

Example The instruction ADDI.L #$1000,D0 adds the hex quantity $1000 (4096 decimal) to all 32 bits of register D0. The immediate data ($1000) is contained in the two words following the instruction, as shown below:

Instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	Size		Mode			Register		
0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0

Data

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

This instruction has hex value $0680 0000 1000. Executing the instruction yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: ffffffff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADDI$       ADDI.L     #$00001000,D0
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2711 -- -- SI PL=7 X---C
D0: 00000fff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADDI$+006   NOP
```

ADDQ Instruction

ADD Quick

Syntax: ADDQ #<data>,<ea>

Description

The ADDQ instruction adds a small immediate operand to an effective address. The immediate data can have the value 1–8. This instruction is two or 4 bytes shorter (depending on data size) than the equivalent ADDI instruction. ADDQ takes advantage of the fact that additions by a small constant are quite common, particularly in code generated by high-level languages. The presence of the ADDQ instruction makes this code smaller and faster than it would ordinarily be.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•	•	•		•	•	•	•

X Set if a carry was generated. Cleared otherwise.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Set if an overflow was generated. Cleared otherwise.
C Set if a carry was generated. Cleared otherwise.

When an address register is used as the destination, all 32 bits of the register are used (regardless of instruction size), and the condition codes are not affected.

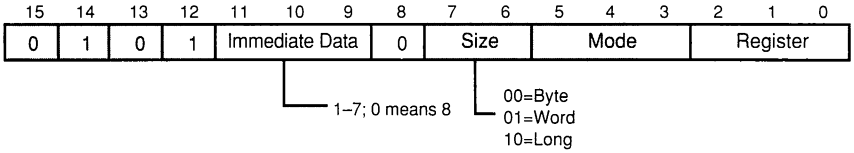
Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•
An *	001	An		•
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100	•	
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

* Word or long only ☐ 68020/030 only

Instruction Format



The immediate data field (bits 11–9) contains the data to be added to the destination. A value of zero in this field is interpreted as 8 (adding zero is not very useful). The size field (bits 7–6) contains 00 for adding bytes, 01 for words, and 10 for longs.

Example The ADDQ.L #1,D1 instruction increments the 32-bit quantity in register D1 by 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	Immediate Data			0	Size		Mode			Register		
0	1	0	1	0	0	1	0	1	0	0	0	0	0	0	1

The hex value of this instruction is $5281. Executing the instruction yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 0000ffff 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADDQ$       ADDQ.L     #$1,D1
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 00010000 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADDQ$+002   NOP
```

ADDX Instruction

ADD Binary with eXtend

Syntax: ADDX Dy,Dx
ADDX -(Ay),-(Ax)

Description

The ADDX (ADD eXtended) instruction permits the addition of quantities that require more than one byte, word, or longword. This instruction is similar to a normal addition, except the eXtend condition code bit serves as a carry from the previous operation. The instruction can have two forms:

1. Add a data register to another data register
2. Add two memory locations using the Address Register Predecrement addressing mode

The 68000 big-endian data format dictates the form of multiple-precision operations with the ADDX instruction. The memory-to-memory form must begin at the highest-addressed byte and work back to the lowest-addressed byte, using predecrement addressing.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•	•	•		•	•	•	•

X Set if a carry was generated. Cleared otherwise.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Cleared if result was non-zero. Unchanged otherwise.
V Set if an overflow was generated. Cleared otherwise.
C Set if a carry was generated. Cleared otherwise.

The behavior of the Z condition code follows the 68000 standard for multiple-precision arithmetic. You normally use a single-precision operation to add the low-order parts of multiple-precision operands. The Z bit will be set if this part of the operation is zero. If it is still set at the end of the operation, the entire result is zero.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Register Rx			1	Size		0	0	R/M	Register Ry		

Size:
00=Byte
01=Word
10=Long

R/M:
0 = ADDX Dy,Dx
1 = ADDX -(Ay),-(Ax)

Bits 11–9 contain the destination register number, and bits 2–0 contain the source register number. Bit 3 (R/M) is zero if the register numbers indicate data registers and one for address registers.

Example This example uses the ADDX instruction to add two pairs of data registers. The operands are each 64 bits (two longwords) in length. The source operand is contained in the D3/D2 register pair, and the destination is contained in the D1/D0 register pair. D2 and D0 contain the low-order portion of the operands.

This example requires two instructions: an ADD instruction to add the low-order 32 bits and an ADDX instruction to add the upper 32 bits. The layout of this latter instruction is

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	Register Rx			1	Size		0	0	R/M	Register Ry		
1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1

The ADDX instruction has hex value $D383. Executing the instructions yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: fffffff0 00000000 00000010 00000000 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADDX$+000   ADD.L      D2,D0

USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2715 -- -- SI PL=7 X-Z-C
D0: 00000000 00000000 00000010 00000000 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADDX$+002   ADDX.L     D3,D1
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 00000000 00000001 00000010 00000000 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ADDX$+006   NOP
```

The ADD instruction adds the two low-order longwords, ignoring the state of the X condition code bit. Note that this instruction correctly sets the Z flag for all subsequent ADDX instructions. Setting the Z flag with a separate instruction is not required. The ADDX adds the high longwords, using the X bit generated by the ADD instruction. Note the carry between the low-order and high-order longwords.

AND Instruction

AND Logical

Syntax: AND <ea>,Dn
AND Dn,<ea>

Description

The AND instruction performs a logical AND instruction between a data register and an effective address operand.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0		•	•	•		•	•	•	•

X Not affected.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Valid Addressing Modes

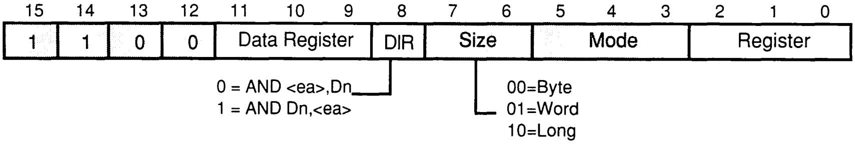
Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	
An	001	An		
(An)	010	An	•	•
(An)+	011	An	•	•
-(An)	100	An	•	•
(d16,An)	101	An	•	•
(d8,An,Xn)	110	An	•	•
(bd,An,Xn)	110	An	•	•
([bd,An,Xn],od)	110	An	•	•
([bd,An],Xn,od)	110	An	•	•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	•
xxx.L	111	001	•	•
#<data>	111	100	•	
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
(bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

If you specify an immediate mode source and a destination mode other than Dn, many assemblers will generate an ANDI instruction.

Instruction Format



Bits 11–9 contain the data register number. Bit 8 indicates the direction. If this bit is set, then the data register field specifies the source operand. If bit 8 is zero, the data register is the destination.

Example The AND.L D0,D1 instruction performs a 32-bit logical AND between the contents of D0 and the contents of D1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Data Register			DIR	Size		Mode			Register		
1	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0

The hex value of the instruction is $C280. Instruction execution yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 98765432 fec80123 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =AND$        AND.L      D0,D1
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2708 -- -- SI PL=7 -N---
D0: 98765432 98400022 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =AND$+002    NOP
```

ANDI Instruction

AND Immediate

Syntax: ANDI #<data>,<ea>

Description

The ANDI instruction performs a logical operation on an effective address operand using immediate data.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0		•	•	•		•	•	•	•

X Not affected.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100	•	
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	Size		Mode			Register		
Word data (16 bits)								Byte data (8 bits)							
Long data (32 bits)															

The size field indicates the operation size: 00 for byte, 01 for word, and 10 for long. There are 2 or 4 bytes of immediate data that follow the instruction in memory. Byte and word operations have 16 bits (2 bytes) of immediate data. Long operations have 32 bits (4 bytes).

Example The instruction ANDI.L #$FEC80123,D0 performs a logical AND between the 32-bit value in register D0 and the hex value $FEC80123. The immediate data is contained in the two words following the instruction, as shown below:

Instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	Size		Mode			Register		
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0

Data

1	1	1	1	1	1	1	0	1	1	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	1

This instruction has hex value $0280 FEC8 0123. Executing the instruction yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 98765432 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ANDI$      ANDI.L     #$FEC80123,D0
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2708 -- -- SI PL=7 -N---
D0: 98400022 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ANDI$+006  NOP
```

ANDI to CCR Instruction

AND Immediate to Condition Code Register

Syntax: ANDI #<data>,CCR

Description

The ANDI to CCR instruction performs a logical AND operation on the Condition Code Register using an 8-bit immediate field. This is useful for clearing any arbitrary combination of condition code bits.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•				•	•	•	•

X Cleared if bit 4 of immediate operand is zero. Unchanged otherwise.
N Cleared if bit 3 of immediate operand is zero. Unchanged otherwise.
Z Cleared if bit 2 of immediate operand is zero. Unchanged otherwise.
V Cleared if bit 1 of immediate operand is zero. Unchanged otherwise.
C Cleared if bit 0 of immediate operand is zero. Unchanged otherwise.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	Byte Data (8 bits)							

Example The ANDI #$0E,CCR instruction clears the X and C condition code bits. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

The instruction has hex value $023C 000E. Execution yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =271f -- -- SI PL=7 XNZVC
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ANDICCR$    ANDI.B     #$0E,CCR
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =270e -- -- SI PL=7 -NZV-
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ANDICCR$+004           NOP
```

ANDI to SR Instruction (Privileged)

Syntax: ANDI #<data>,SR

AND Immediate to Status Register

Description

The ANDI to SR instruction performs a logical AND operation on the Status Register using an 16-bit immediate field. This is useful for clearing any arbitrary combination of bits in this register. The processor must be in Supervisor mode to execute this instruction. Attempting execution in user mode causes a Privilege Violation exception through vector 8.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*			•			•	•	•	•

X Cleared if bit 4 of immediate operand is zero. Unchanged otherwise.

N Cleared if bit 3 of immediate operand is zero. Unchanged otherwise.

Z Cleared if bit 2 of immediate operand is zero. Unchanged otherwise.

V Cleared if bit 1 of immediate operand is zero. Unchanged otherwise.

C Cleared if bit 0 of immediate operand is zero. Unchanged otherwise.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0
Word Data (16 bits)															

Example The ANDI #$2413,SR instruction clears the N and Z condition code bits, as well as both trace bits, the Master status bit, and the low 2 bits of the interrupt mask. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0
0	0	1	0	0	1	0	0	0	0	0	1	0	0	1	1

The instruction has hex value $027C 2413. Execution yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =271f -- -- SI PL=7 XNZVC
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ANDISR$     ANDI.W     #$2413,SR
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2413 -- -- SI PL=4 X--VC
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ANDISR$+004            NOP
```

ASL Instruction

Arithmetic Shift Left

Syntax:

```
ASL      Dx,Dy
ASL      #<data>,Dy
ASL      <ea>
```

Description

The ASL instruction performs an arithmetic shift left on a data register or on an effective address operand. The shift count for a data register can be specified one of two ways: The low 5 bits of a data register can be used for the count value (ASL Dx,Dy form), or a small constant in the instruction itself can hold the count (ASL #<data>,Dy form). The register form allows shifts of up to 63 bits, but the immediate constant can have a value in the range 1–8. The data register form of the instruction supports all sizes — byte, word, and longword.

The effective address form (ASL <ea>) shifts only 1 bit and supports only the word data size.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•	•	•		•	•	•	•

☐ Register form only

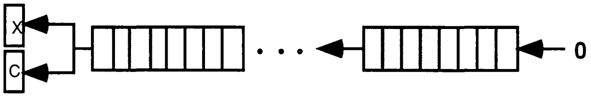
X Set to the last bit shifted out of the operand. Unaffected for a shift count of zero.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Set if the most significant bit changes at any time during the operation. Cleared otherwise.
C Set to the last bit shifted out of the operand. Cleared for a shift count of zero.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100	•	
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only ☐ Register form only

Bits shifted out of the most significant bit (31, 15, or 7) are placed in both the Extend (X) and Carry (C) condition code bits, as shown below. Zeros are shifted in from the right. This operation is effectively a multiplication by a power of two (the power is the number of bits shifted).



Instruction Format

Register Shifts

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	Count/Register	1	Size	i/r	0	0	Register

Size:
00=Byte
01=Word
10=Long

i/r:
0 = Immediate Shift count
1 = Register Shift count

The Count/Register field contains either a register number (bit 5=1) or an immediate shift count (bit 5 = 0). The immediate shift count form uses bits 11–9 to hold a value from 1–7. A value of zero in these bits represents a value of 8. Bits 2–0 contain the number of the register to be shifted.

Memory Shifts

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	1	1	1	Mode			Register		

Bit 8 in both formats indicates the shift direction. A one indicates a left shift, and zero indicates a right shift.

Example The ASL.L D1,D0 instruction shifts the 32-bit quantity in D0 left by the number of bits specified by register D1. This instruction has the following format:

1	1	1	0	Count/Register			1	Size		i/r	0	0	Register		
1	1	1	0	0	0	1	1	1	0	1	0	0	0	0	0

The instruction has hex value $E3A0. Execution yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 87654321 00000013 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ASL$        ASL.L      D1,D0
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2702 -- -- SI PL=7 ---V-
D0: 19080000 00000013 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ASL$+002    NOP
```

In this example, the shift count is $13 (19 decimal). The pattern chosen for D0 caused the sign to change, thereby setting the V condition code bit.

ASR Instruction

Arithmetic Shift Right

Syntax:
ASR Dx,Dy
ASR #<data>,Dy
ASR <ea>

Description

The ASR instruction performs an arithmetic shift right on a data register or on an effective address operand. The shift count for a data register can be specified one of two ways: The low 5 bits of a data register can be used for the count value (ASR Dx,Dy form), or a small constant in the instruction itself can hold the count (ASR #<data>,Dy form). The register form allows shifts of up to 63 bits, while the immediate constant can have a value in the range 1–8. The data register form of the instruction supports all sizes — byte, word, and longword.

The effective address form (ASR <ea>) shifts only 1 bit and supports only the word data size.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		▨ ●	●	●		●	●	●	●

▨ Register form only

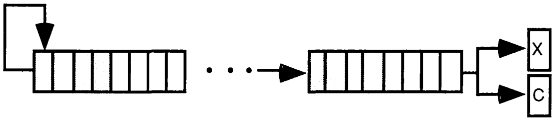
X Set to the last bit shifted out of the operand. Unaffected for a shift count of zero.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Set if the most significant bit changes at any time during the operation. Cleared otherwise.
C Set to the last bit shifted out of the operand. Cleared for a shift count of zero.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100	•	
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only ☐ Register form only

Bits shifted out of bit 0 are placed in both the Extend (X) and Carry (C) condition code bits, as shown below. The most significant bit (31, 15, or 7) is replicated on the left so that the original sign of the number is preserved. This operation is effectively a signed division by a power of two (the power is the number of bits shifted) for all numbers except -1 (which remains -1 when shifted).



Instruction Format

Register Shifts

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			0	Size		i/r	0	0	Register		

00=Byte
01=Word
10=Long

0 = Immediate Shift count
1 = Register Shift count

The Count/Register field contains either a register number (bit 5 = 1) or an immediate shift count (bit 5 = 0). The immediate shift count form uses bits 11–9 to hold a value from 1–7. A value of zero in these bits represents a value of 8. Bits 2–0 contain the number of the register to be shifted.

Memory Shifts

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	0	1	1	Mode			Register		

Bit 8 in both formats indicates the shift direction. A one indicates a left shift, and zero indicates a right shift.

Example The ASR.L D1,D0 instruction shifts the 32-bit quantity in D0 right by the number of bits specified by register D1. This instruction has the following format:

1	1	1	0	Count/Register			0	Size		i/r	0	0	Register		
1	1	1	0	0	0	1	0	1	0	1	0	0	0	0	0

The instruction has hex value $E2A0. Execution yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 87654321 00000013 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ASR$        ASR.L      D1,D0
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2719 -- -- SI PL=7 XN--C
D0: fffff0ec 00000013 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =ASR$+002    NOP
```

In this example, the shift count is $13 (19 decimal). The pattern chosen for D0 was negative to begin with, resulting in 19 one bits being shifted into the top part of the register. The N condition code bit became set as a result. The last bit shifted out of bit 0 was a one, which sets both the X and C condition code bits.

Bcc Instruction

Branch Conditionally

Syntax: Bcc <label>

Description

The Bcc instructions cause a conditional branch depending on the settings of the low-order 4 bits in the Condition Code Register (CCR).

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-		•	•	•		•	•	•	•

☐ 68020/030 only

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	Condition				8-bit displacement							
16-bit displacement if 8-bit displacement = $00															
32-bit displacement if 8-bit displacement = $FF (68020/68030 only)															

The branch instructions conditionally alter the value of the program counter (PC) register depending on the setting of the condition codes. The instruction can be 2, 4, or 6 bytes (on the 68020/030) in length. The 8-bit form uses the low-order byte of the instruction word as a signed displacement to be added to the program counter. If this 8-bit quantity is zero, the next word is used as the displacement. If the 8-bit quantity is $FF (255 decimal), the next longword following the instruction is used as the displacement. A branch to the next instruction (8-bit offset is zero) requires the 16-bit form. The value $FF does not create a problem since it is an odd value, and instruction addresses must be even. This value does not conflict with any valid 2–byte branch instruction. The value used for the program counter is always the word following the first instruction word (the instruction address plus 2).

Conditions The condition field (bits 11–8) indicates the combination of condition codes desired. Four bits allow sixteen possible combinations. Since one of these combinations, 0001, is used for the BSR (Branch to SubRoutine) instruction, there are actually only fifteen combinations. The conditions and the corresponding instructions in the BCC family follow.

Condition	**Meaning**
0000 ($0)	**BRA** (**BR**anch **A**lways). This instruction always branches.
0001 ($1)	This condition code is used for the **BSR** (Branch to SubRoutine) instruction.
0010 ($2)	**BHI** (**B**ranch **HI**gher). This instruction branches if both the Carry and Zero condition code bits are clear. BHI is equivalent to Branch if Greater than (**BGT**) for unsigned numbers.
0011 ($3)	**BLS** (**B**ranch **L**ower or **S**ame). This instruction branches if either the Carry or Zero condition code bits are set. BLS is equivalent to Branch if Less than or Equal (**BLE**) for unsigned numbers.
0100 ($4)	**BCC** (**B**ranch if **C**arry **C**lear). This instruction branches if the Carry bit is not set.
0101 ($5)	**BCS** (**B**ranch if **C**arry **S**et). This instruction branches if the Carry bit is set.
0110 ($6)	**BNE** (**B**ranch if **N**ot **E**qual). This instruction branches if the Z bit is not set.
0111 ($7)	**BEQ** (**B**ranch if **EQ**ual). This instruction branches if the Z bit is set.
1000 ($8)	**BVC** (**B**ranch if o**V**erflow **C**lear). This instruction branches if the V bit is not set.
1001 ($9)	**BVS** (**B**ranch if o**V**erflow **S**et). This instruction branches if the V bit is set.
1010 ($A)	**BPL** (**B**ranch if **PL**us). This instruction branches if the N bit is not set.
1011 ($B)	**BMI** (**B**ranch if **MI**nus). This instruction branches if the N bit is set.
1100 ($C)	**BGE** (**B**ranch if **G**reater than or **E**qual). This instruction branches if the N and V bits are both set or if the N and V bits are both clear.
1101 ($D)	**BLT** (**B**ranch if **L**ess **T**han). This instruction branches if the N bit is set and the V bit is clear, or if the N bit is clear and the V bit is set.
1110 ($E)	**BGT** (**B**ranch if **G**reater **T**han). This instruction branches if the N and V bits are both set and the Z bit is clear, or if the N, V, and Z bits are all clear.
1111 ($F)	**BLE** (**B**ranch if **L**ess Than or **E**qual). This instruction branches if the Z bit is set, if the N bit is set and the V bit is clear, or if the N bit is clear and the V bit is set.

Example The BEQ *+$000A instruction branches forward if the Z bit in the Condition Code Register is set. (The assembly language symbol * refers to the location of the first word of the instruction.) This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	Condition				8-Bit Displacement							
0	1	1	0	0	1	1	1	0	0	0	0	1	0	0	0

The instruction has hex value $6708 (the 2-byte form). Execution yields:

Before:

```
PC =BCC$          BEQ          *+$000A       ; TAKEN
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2704 -- -- SI PL=7 --Z--
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2704 -- -- SI PL=7 --Z--
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =TAKEN         NOP
```

Only the value in the Program Counter (PC) changes as a result of execution.

BCHG Instruction

Test a Bit and Change (invert)

Syntax: BCHG Dn,<ea>
BCHG #<data>,<ea>

Description

The BCHG instruction tests a bit in the destination. The bit is then complemented. The Z bit in the condition code register reflects the state of the bit before being complemented. The bit number can be specified in a data register or in an immediate field in the instruction itself. A data register destination implies a long (32-bit) data size. All other destination operands imply a byte (8-bit) data size.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	*	-	-		•		•		•	•	•	•

☐ Data Register Destination only

X Not affected.
N Not affected.
Z Set if the bit tested was zero before the instruction began. Cleared otherwise.
V Not affected.
C Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000		•
An	001	An		•	xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100	•	
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only ☐ Long data size only

Instruction Format

Bit Number Dynamic

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register Dn			1	0	1	Mode			Register		

The Register Dn field contains the number of the data register whose low-order 5 bits (long-data size) or 3 bits (byte-data size) specify the bit number to be inverted in the destination operand.

Bit Number Static

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	Mode			Register		
0	0	0	0	0	0	0	0	Bit Number							

Only bits 5–0 (long-data size) or bits 2–0 (byte-data size) of the bit number operand have meaning. All other bits are ignored.

Example The BCHG #$02,D0 instruction inverts the value of bit 2 of data register D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	Mode			Register		
0	0	0	0	0	0	0	0	Bit Number							

0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

The instruction has hex value $0840 0002. Execution yields:

Before

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 00000000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =BCHG$       BCHG       #$02,D0
```

After

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2704 -- -- SI PL=7 --Z--
D0: 00000004 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =BCHG$+004   NOP
```

BCLR Instruction

Test a Bit and Clear

Syntax: BCLR Dn,<ea>
BCLR #<data>,<ea>

Description

The BCLR instruction tests a bit in the destination. The bit is then cleared. The Z bit in the condition code register reflects the state of the bit before being cleared. The bit number can be specified in a data register or in an immediate field in the instruction itself. The destination can be a data register or an effective address operand. A data register destination implies a long (32-bit) data size. All other destination operands imply a byte (8-bit) data size.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-		•		•		•	•	•	•

☐ Data Register Destination only

X Not affected.
N Not affected.
Z Set if the bit cleared was zero before the instruction began. Cleared otherwise.
V Not affected.
C Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000		•
An	001	An		•	xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100	•	
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only ☐ Long data size only

Instruction Format

Bit Number Dynamic

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register Dn			1	1	0	Mode			Register		

The Register Dn field contains the number of the data register whose low-order 5 bits (long-data size) or 3 bits (byte-data size) specify the bit number to be cleared in the destination operand.

Bit Number Static

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	Mode			Register		
0	0	0	0	0	0	0	0	Bit Number							

The bit number is modulo 32. This means that only bits 5–0 (long-data size) or bits 2–0 (byte-data size) of the bit number operand have meaning. All others are ignored.

Example The BCLR #$05,D0 instruction clears bit 5 of data register D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	Mode			Register		
0	0	0	0	0	0	0	0	Bit Number							

0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

The instruction has hex value $0880 0005. Execution yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: ffffffff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =BCLR$       BCLR       #$05,D0
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: ffffffdf 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =BCLR$+004   NOP
```

Note that no condition code bits changed because the bit was set before the instruction.

BFCHG Instruction

Test Bit Field and Change

Syntax: BFCHG <ea>{offset:width}

Description

The BFCHG instruction inverts a bit field at the designated effective address. The condition codes are set according to the bit field value before the instruction. You specify a bit field using an offset and a width. The offset is the number of bits to the left of the field. The width is the number of bits in the field.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0								•	•

X Not affected.

N Set if bit field was negative (highest bit in field = 1). Cleared otherwise.

Z Set if bit field was zero (all bits in field = 0). Cleared otherwise.

V Always cleared.

C Always cleared.

The condition codes are set according to the value of the bit field before the instruction began. This instruction works on variable-length data (bit fields can be from 1–32 bits in length). Bit fields are not supported on the 68000 and 68010.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•
An	001	An		
(An)	010	An		•
(An)+	011	An		
-(An)	100	An		
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100		
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

This instruction has no source operand.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	1	1	Mode			Register		
0	0	0	0	Do	Offset					Dw	Width				

Do: 0 = Bits 10–6 contain offset
1 = Bits 8–6 specify data register

Dw: 0 = Bits 4–0 contain width
1 = Bits 2–0 specify data register

The offset and width fields can be contained in data registers or in the offset (bits 10–6) and width (bits 4–0) fields in the second word of the instruction. Bits Do (11) and Dw (5) are zero to indicate immediate fields and one to indicate data registers.

The offset field can have values from 0–31 if specified as an immediate operand. If the offset field is specified in a data register, Do is a one, and bits 8–6 contain the data register number. Using a data register, the offset value can have a full 32-bit range of -2^{31} to $2^{31}-1$.

If the width field is specified in a data register, Dw is a one, and bits 2–0 contain the data register number. The low-order 5 bits contain the value required in both cases. A value of zero signifies a width of 32.

Example The BFCHG D0{12:4} instruction inverts bits 19–16 (12 bits from the left for 4 bits, or the fourth hex digit from the left). This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	1	1	Mode			Register		
0	0	0	0	Do	Offset					Dw	Width				

1	1	1	0	1	0	1	0	1	1	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0

The instruction has the hex value $EAC0 0304. Execution yields:

Before:

```
USP=b03f8620 MSP=b03f8620 ISP =b03f8620 SR =2700 -- -- SI PL=7 -----
D0: 01234567 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8620
PC =BFCHG$      BFCHG      D0{$0C:$04}
```

After:

```
USP=b03f8620 MSP=b03f8620 ISP =b03f8620 SR =2700 -- -- SI PL=7 -----
D0: 012c4567 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8620
PC =BFCHG$+004  NOP
```

BFCLR Instruction

Bit Field Clear

Syntax: BFCLR <ea>{offset:width}

Description

The BFCLR instruction clears a bit field at the designated effective address. The condition codes are set according to the bit field value before the instruction. You specify a bit field using an offset and a width. The offset is the number of bits to the left of the field. The width is the number of bits in the field.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0								•	•

X Not affected.
N Set if bit field was negative (highest bit in field = 1). Cleared otherwise.
Z Set if bit field was zero (all bits in field = 0). Cleared otherwise.
V Always cleared.
C Always cleared.

The condition codes are set according to the value of the bit field before the instruction began. This instruction works on variable-length data (bit fields can be from 1–32 bits in length). Bit fields are not supported on the 68000 and 68010.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100		
(An)+	011	An							
-(An)	100	An							
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

This instruction has no source operand.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	1	1	Mode			Register		
0	0	0	0	Do	Offset					Dw	Width				

0 = Bits 10–6 contain offset
1 = Bits 8–6 specify data register

0 = Bits 4–0 contain width
1 = Bits 2–0 specify data register

The offset and width fields can be contained in data registers or in the offset (bits 10–6) and width (bits 4–0) fields in the second word of the instruction. Bits Do (11) and Dw (5) are zero to indicate immediate fields and one to indicate data registers.

The offset field can have values from 0–31 if specified as an immediate operand. If the offset field is specified in a data register, Do is a one, and bits 8–6 contain the data register number. Using a data register, the offset value can have a full 32-bit range: -2^{31} to $2^{31}-1$.

If the width field is specified in a data register, Dw is a one, and bits 2–0 contain the data register number. The low-order 5 bits contain the value required in both cases. A value of zero signifies a width of 32.

Example The BFCLR D0{12:4} instruction clears bits 19–16 (12 bits from the left for 4 bits, or the fourth hex digit from the left). This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	1	1	Mode			Register		
0	0	0	0	Do	Offset					Dw	Width				

1	1	1	0	1	1	0	0	1	1	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0

The instruction has the hex value $ECC0 0304. Execution yields:

Before:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2700 -- -- SI PL=7 -----
D0: 01234567 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =BFCLR$      BFCLR      D0{$0C:$04}
```

After:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2700 -- -- SI PL=7 -----
D0: 01204567 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =BFCLR$+004  NOP
```

BFEXTS Instruction

Syntax: BFEXTS <ea>{offset:width},Dn

Bit Field Extract Signed (<ea> to Data Register)

Description

The BFEXTS instruction copies a bit field at the designated effective address into a data register. The condition codes are set according to the bit field value. You specify a bit field using an offset and a width. The offset is the number of bits to the left of the field. This value specifies the most significant bit in the field. The width is the number of bits in the field. The bit field is placed in the low-order bits of the data register. The high-order bit of the bit field is extended into the high-order bits of the data register.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0								•	•

X Not affected.
N Set if bit field is negative (highest bit in field = 1). Cleared otherwise.
Z Set if bit field is zero (all bits in field = 0). Cleared otherwise.
V Always cleared.
C Always cleared.

This instruction works on variable-length data (bit fields can be from 1–32 bits in length). Bit fields are not supported on the 68000 and 68010.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An	•	
(An)+	011	An		
-(An)	100	An		
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	110	An	•	
([bd,An],Xn,od)	110	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100		
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	1	Mode			Register		
0	Dest. Register			Do	Offset					Dw	Width				

Do: 0 = Bits 10–6 contain offset
1 = Bits 8–6 specify data register

Dw: 0 = Bits 4–0 contain width
1 = Bits 2–0 specify data register

Bits 14–12 of the extension word specify the destination data register. The offset and width fields can be contained in data registers or in the offset (bits 10–6) and width (bits 4–0) fields in the second word of the instruction. Bits Do (11) and Dw (5) are zero to indicate immediate fields and one to indicate data registers.

The offset field can have values from 0–31 if specified as an immediate operand. If the offset field is specified in a data register, Do is a one, and bits 8–6 contain the data register number. Using a data register, the offset value can have a full 32-bit range: -2^{31} to $2^{31}-1$.

If the width field is specified in a data register, Dw is a one, and bits 2–0 contain the data register number. The low-order 5 bits contain the value required in both cases. A value of zero signifies a width of 32.

Example The BFEXTS D0{12:4},D1 instruction places bits 19–16 from D0 into the low four bits of D1. Bit 19 from D0 is copied into all other bits of D1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	1	Mode			Register		
0	Dest. Register			Do	Offset					Dw	Width				

1	1	1	0	1	0	1	1	1	1	0	0	0	0	0	0
0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	0

The hex value of the instruction is $EBC0 1304. Execution yields:

Before:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2700 -- -- SI PL=7 -----
D0: 89abcdef 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =BFEXTS$     BFEXTS     D0{$0C:$04},D1
```

After:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2708 -- -- SI PL=7 -N---
D0: 89abcdef fffffffb 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =BFEXTS$+004            NOP
```

The sign extension is present because the hex digit $B has the high-order bit set. This also caused the N condition code bit to be set.

BFEXTU Instruction Syntax: BFEXTU <ea>{offset:width},Dn

Bit Field Extract Unsigned (<ea> to Data Register)

Description

The BFEXTU instruction copies a bit field at the designated effective address into a data register. The condition codes are set according to the bit field value. You specify a bit field using an offset and a width. The offset is the number of bits to the left of the field. This value specifies the most significant bit in the field. The width is the number of bits in the field. The bit field is placed in the low-order bits of the data register. The high-order bits of the data register are cleared.

Condition Codes	X	N	Z	V	C	**Sizes**	Byte	Word	Long	**CPUs**	000	010	020	030
	-	*	*	0	0								•	•

X Not affected.
N Set if bit field is negative (highest bit in field = 1). Cleared otherwise.
Z Set if bit field is zero (all bits in field = 0). Cleared otherwise.
V Always cleared.
C Always cleared.

This instruction works on variable-length data (bit fields can be from 1–32 bits in length). Bit fields are not supported on the 68000 and 68010.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An	•	
(An)+	011	An		
-(An)	100	An		
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	110	An	•	
([bd,An],Xn,od)	110	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100		
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	1	1	Mode			Register		
0	Dest. Register			Do	Offset					Dw	Width				

Do: 0 = Bits 10–6 contain offset
1 = Bits 8–6 specify data register

Dw: 0 = Bits 4–0 contain width
1 = Bits 2–0 specify data register

Bits 14–12 of the extension word specify the destination data register. The offset and width fields can be contained in data registers or in the offset (bits 10–6) and width (bits 4–0) fields in the second word of the instruction. Bits Do (11) and Dw (5) are zero to indicate immediate fields and one to indicate data registers.

The offset field can have values from 0–31 if specified as an immediate operand. If the offset field is specified in a data register, Do is a one, and bits 8-6 contain the data register number. Using a data register, the offset value can have a full 32-bit range: -2^{31} to $2^{31}-1$.

If the width field is specified in a data register, Dw is a one, and bits 2–0 contain the data register number. The low-order 5 bits contain the value required in both cases. A value of zero signifies a width of 32.

Example The BFEXTU D0{12:4},D1 instruction places bits 19–16 from D0 into the low 4 bits of D1. All other bits of D1 are set to zero. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	1	1	Mode			Register		
0	Dest. Register			Do	Offset					Dw	Width				

1	1	1	0	1	0	0	1	1	1	0	0	0	0	0	0
0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	0

The hex value of the instruction is $E9C0 1304. Execution yields:

Before:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2700 -- -- SI PL=7 -----
D0: 89abcdef 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =BFEXTU$     BFEXTU     D0{$0C:$04},D1
```

After:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2708 -- -- SI PL=7 -N---
D0: 89abcdef 0000000b 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =BFEXTU$+004            NOP
```

Note that the N condition code bit was set, indicating that the high-order bit of the bit field was a one.

BFFFO Instruction

Syntax: BFFFO <ea>{offset:width},Dn

Bit Field Find First One

Description

The BFFFO instruction finds the most significant bit, which has the value one in a bit field at the designated effective address. The condition codes are set according to the bit field value. You specify a bit field using an offset and a width. The offset is the number of bits to the left of the field. This value specifies the most significant bit in the field. The width is the number of bits in the field. The destination data register is set to the original offset plus the offset within the field of the first bit number, which is set to one. If the bit field is all zeros, the destination data register is set to the offset value plus the bit field width. (The Z condition code bit will also be set in this case.)

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0								•	•

X Not affected.
N Set if bit field is negative (highest bit in field = 1). Cleared otherwise.
Z Set if bit field is zero (all bits in field = 0). Cleared otherwise.
V Always cleared.
C Always cleared.

This instruction works on variable-length data (bit fields can be from 1–32 bits in length). Bit fields are not supported on the 68000 and 68010.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000	•	
An	001	An			xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100		
(An)+	011	An							
-(An)	100	An							
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	1	1	Mode			Register		
0	Dest. Register			Do	Offset					Dw	Width				

0 = Bits 10–6 contain offset
1 = Bits 8–6 specify data register

0 = Bits 4–0 contain width
1 = Bits 2–0 specify data register

Bits 14–12 of the extension word specify the destination data register. The offset and width fields can be contained in data registers or in the offset (bits 10–6) and width (bits 4–0) fields in the second word of the instruction. Bits Do (11) and Dw (5) are zero to indicate immediate fields and one to indicate data registers.

The offset field can have values from 0–31 if specified as an immediate operand. If the offset field is specified in a data register, Do is a one, and bits 8-6 contain the data register number. Using a data register, the offset value can have a full 32-bit range: -2^{31} to 2^{31}-1.

If the width field is specified in a data register, Dw is a one, and bits 2–0 contain the data register number. The low-order 5 bits contain the value required in both cases. A value of zero signifies a width of 32.

Example The BFFFO D0{0:0},D1 instruction loads the bit number (starting at zero from the left) of the first one bit in D0 into D1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	1	1	1	Mode			Register		
0	Dest. Register			Do	Offset					Dw	Width				

1	1	1	0	1	1	0	1	1	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

The instruction has hex value $EDC0 1000. Execution yields:

Before:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2700 -- -- SI PL=7 -----
D0: 00030000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =BFFFO$      BFFFO      D0{$00:$00},D1
```

After:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2700 -- -- SI PL=7 -----
D0: 00030000 0000000e 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =BFFFO$+004  NOP
```

The leftmost one bit in D0 (value $3000) is bit 17 (counting from the right). Counting from the left, this same bit is bit 14 (31–17, or $E in D1 after the instruction).

BFINS Instruction **Syntax: BFINS Dn,<ea>{offset:width}**

Bit Field Insert (Data Register to <ea>)

Description

The BFINS instruction copies a bit field in the low-order bits of a data register to a bit field at the designated effective address. The condition codes are set according to the bit field value. You specify a bit field using an offset and a width. The offset is the number of bits to the left of the field. This value specifies the most significant bit in the field. The width is the number of bits in the field.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0								•	•

X	Not affected.
N	Set if bit field is negative (highest bit in field = 1). Cleared otherwise.
Z	Set if bit field is zero (all bits in field = 0). Cleared otherwise.
V	Always cleared.
C	Always cleared.

This instruction works on variable-length data (bit fields can be from 1–32 bits in length). Bit fields are not supported on the 68000 and 68010.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100		
(An)+	011	An							
-(An)	100	An							
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	1	Mode			Register		
0	Dest. Register			Do	Offset					Dw	Width				

Do: 0 = Bits 10–6 contain offset
1 = Bits 8–6 specify data register

Dw: 0 = Bits 4–0 contain width
1 = Bits 2–0 specify data register

Bits 14–12 of the extension word specify the destination data register. The offset and width fields can be contained in data registers or in the offset (bits 10–6) and width (bits 4–0) fields in the second word of the instruction. Bits Do (11) and Dw (5) are zero to indicate immediate fields and one to indicate data registers.

The offset field can have values from 0–31 if specified as an immediate operand. If the offset field is specified in a data register, Do is a one, and bits 8–6 contain the data register number. Using a data register, the offset value can have a full 32-bit range: -2^{31} to $2^{31}-1$.

If the width field is specified in a data register, Dw is a one, and bits 2–0 contain the data register number. The low-order 5 bits contain the value required in both cases. A value of zero signifies a width of 32.

Example The BFINS D1,D0{8:4} instruction places the low-order 4 bits in register D1 into bits 23-20 of register D0. This field is the third hexadecimal digit from the left. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	1	Mode			Register		
0	Dest. Register			Do	Offset					Dw	Width				

1	1	1	0	1	1	1	1	1	1	0	0	0	0	0	0
0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0

The instruction has hex value $EFC0 1204. Execution yields:

Before:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2700 -- -- SI PL=7 -----
D0: 890bcdef 0000000a 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =BFINS$      BFINS      D1,D0{$08:$04}
```

After:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2708 -- -- SI PL=7 -N---
D0: 89abcdef 0000000a 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =BFINS$+004  NOP
```

BFSET Instruction

Syntax: BFSET <ea>{offset:width}

Bit Field Set

Description

The BFSET instruction sets the bit field at the designated effective address to all ones. The condition codes are set according to the bit field value before the instruction. You specify a bit field using an offset and a width. The offset is the number of bits to the left of the field. This value specifies the most significant bit in the field. The width is the number of bits in the field.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0								•	•

X Not affected.
N Set if bit field was negative (highest bit in field = 1). Cleared otherwise.
Z Set if bit field was zero (all bits in field = 0). Cleared otherwise.
V Always cleared.
C Always cleared.

This instruction works on variable-length data (bit fields can be from 1–32 bits in length). Bit fields are not supported on the 68000 and 68010.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100		
(An)+	011	An							
-(An)	100	An							
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

This instruction has no source operand. The operand can only be written.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	Mode			Register		
0	0	0	0	Do	Offset					Dw	Width				

0 = Bits 10–6 contain offset
1 = Bits 8–6 specify data register

0 = Bits 4–0 contain width
1 = Bits 2–0 specify data register

The offset and width fields can be contained in data registers or in the offset (bits 10–6) and width (bits 4–0) fields in the second word of the instruction. Bits Do (11) and Dw (5) are zero to indicate immediate fields and one to indicate data registers.

The offset field can have values from 0–31 if specified as an immediate operand. If the offset field is specified in a data register, Do is a one, and bits 8-6 contain the data register number. Using a data register, the offset value can have a full 32-bit range: -2^{31} to $2^{31}-1$.

If the width field is specified in a data register, Dw is a one, and bits 2–0 contain the data register number. The low-order 5 bits contain the value required in both cases. A value of zero signifies a width of 32.

Example The BFSET D0{8:4} instruction sets bits 23–20 of register D0. Bits 23–20 constitute the third hex digit from the left. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	Mode			Register		
0	0	0	0	Do	Offset					Dw	Width				

1	1	1	0	1	1	1	0	1	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0

The instruction has hex value $EEC0 0204. Execution yields:

Before:

```
USP=b03f8620 MSP=b03f8620 ISP =b03f8620 SR =2700 -- -- SI PL=7 -----
D0: 890bcdef 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8620
PC =BFSET$      BFSET      D0{$08:$04}
```

After:

```
USP=b03f8620 MSP=b03f8620 ISP =b03f8620 SR =2704 -- -- SI PL=7 --Z--
D0: 89fbcdef 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8620
PC =BFSET$+004  NOP
```

BFTST Instruction

Syntax: BFTST <ea>{offset:width}

Bit Field Test

Description

The BFTST instruction sets the condition codes according to the bit field at the designated effective address. You specify a bit field using an offset and a width. The offset is the number of bits to the left of the field. This value specifies the most significant bit in the field. The width is the number of bits in the field.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0								•	•

X Not affected.
N Set if bit field is negative (highest bit in field = 1). Cleared otherwise.
Z Set if bit field is zero (all bits in field = 0). Cleared otherwise.
V Always cleared.
C Always cleared.

This instruction works on variable-length data (bit fields can be from 1–32 bits in length). Bit fields are not supported on the 68000 and 68010.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•		xxx.W	111	000	•	
An	001	An			xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100		
(An)+	011	An							
-(An)	100	An							
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

This instruction has no destination operand. The effective address is only read.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	Mode			Register		
0	0	0	0	Do	Offset					Dw	Width				

Do: 0 = Bits 10–6 contain offset
1 = Bits 8–6 specify data register

Dw: 0 = Bits 4–0 contain width
1 = Bits 2–0 specify data register

The offset and width fields can be contained in data registers or in the offset (bits 10–6) and width (bits 4–0) fields in the second word of the instruction. Bits Do (11) and Dw (5) are zero to indicate immediate fields and one to indicate data registers.

The offset field can have values from 0–31 if specified as an immediate operand. If the offset field is specified in a data register, Do is a one, and bits 8–6 contain the data register number. Using a data register, the offset value can have a full 32-bit range: -2^{31} to $2^{31}-1$.

If the width field is specified in a data register, Dw is a one, and bits 2–0 contain the data register number. The low-order 5 bits contain the value required in both cases. A value of zero signifies a width of 32.

Example The BFTST D0{8:4} instruction sets the condition codes according to bits 23–20 (the third hex digit from the left) in register D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	Mode			Register		
0	0	0	0	Do	Offset					Dw	Width				

1	1	1	0	1	0	0	0	1	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0

The instruction has hex value $E8C0 0204. Execution yields:

Before:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2700 -- -- SI PL=7 -----
D0: 890bcdef 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =BFTST$      BFTST      D0{$08:$04}
```

After:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2704 -- -- SI PL=7 --Z--
D0: 890bcdef 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =BFTST$+004  NOP
```

Since the selected bits are zero, the Z condition code bit is set after the instruction.

BKPT Instruction

Syntax: BKPT #<data>

Breakpoint

Description

The BKPT instruction causes a breakpoint acknowledge cycle. External debugging hardware (usually an in-circuit emulator) can supply an instruction that replaces the BKPT instruction in the execution stream. The instruction has a 3-bit immediate field that can be used for a breakpoint number. This 3-bit field is placed on address lines A4–A2 during the breakpoint acknowledge cycle. External hardware may respond with an instruction to replace the BKPT instruction or with a BUSERR signal. A BUSERR signal or lack of external hardware causes an Illegal Instruction exception through vector 4.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-								•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

This instruction has no data size and is not supported on the 68000 and 68010.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	1	Immediate		

The immediate field is placed on the address lines during the breakpoint acknowledge cycle. External debugging hardware is thus limited to eight breakpoints.

Example When executed on a machine with no debugging hardware, the BKPT #0 instruction yields an illegal instruction exception. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	1	Immediate		
0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0

This instruction has the hex value $4848. Execution yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =BKPT$      BKPT      #$0
```

After:

```
Illegal Instruction
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =a700 T1 -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =BKPT$      BKPT      #$0
```

Note that the program counter does not advance for an illegal instruction. The status display shows that the illegal instruction exception was taken before the trace exception. (The trace bit T1 is set after the exception.)

BSET Instruction

Test a Bit and Set

Syntax: **BSET Dn,<ea>**
BSET #<data>,<ea>

Description

The BSET instruction tests a bit in the destination. The bit is then set. The Z bit in the condition code register reflects the state of the bit before being set. The bit number can be specified in a data register or in an immediate field in the instruction itself. The destination can be a data register or an effective address operand. A data register destination implies a long (32-bit) data size. All other destination operands imply a byte (8-bit) data size.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	*	-	-		•		•		•	•	•	•

X Not affected.
N Not affected.
Z Set if the bit tested was zero before the instruction began. Cleared otherwise.
V Not affected.
C Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000		•
An	001	An		•	xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100	•	
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only ☐ Long data size only

Instruction Format

Bit Number Dynamic

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register Dn			0	1	1	Mode			Register		

The Register Dn field contains the number of the data register whose low-order 5 bits (long-data size) or 3 bits (byte-data size) specify the bit number in the destination operand.

Bit Number Static

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1		Mode		Register		
0	0	0	0	0	0	0	0	Bit Number							

The bit number is modulo 32. This means that only bits 5–0 (long-data size) or bits 2–0 (byte-data size) of the bit number operand have meaning. All other bits are ignored.

Example The BSET #$07,D0 instruction sets bit 7 of data register D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1		Mode		Register		
0	0	0	0	0	0	0	0	Bit Number							

0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

The instruction has hex value $08C0 0007. Execution yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 00000001 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =BSET$       BSET       #$07,D0
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2704 -- -- SI PL=7 --Z--
D0: 00000081 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =BSET$+004   NOP
```

BSR Instruction

Syntax: BSR <label>

Branch to Subroutine

Description

The BSR instruction pushes the address of the next instruction on the A7 stack and adds the offset contained in the instruction to the program counter. A subsequent RTS (Return from Subroutine) instruction branches to the instruction following the BSR.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-		•	•	•		•	•	•	•

☐ 68020/030 only

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	1	8-bit displacement							
16-bit displacement if 8-bit displacement = $00															
32-bit displacement if 8-bit displacement = $FF (68020/68030 only)															

The instruction can be 2, 4, or 6 bytes (on the 68020/030) in length. The 8-bit form uses the low-order byte of the instruction word as a signed displacement to be added to the program counter. If this 8-bit quantity is zero, the next word is used as the displacement. If the 8-bit quantity is $FF (255 decimal), the next longword following the instruction is used as the displacement. A branch to the next instruction (8-bit offset is zero) requires the 16-bit form. The value $FF does not create a problem since it is an odd value and instruction addresses must be even. This value does not conflict with any valid 2–byte BSR instruction. The value used for the program counter is always the word following the first instruction word (that is, the instruction address plus 2).

Note that the BSR instruction is a Bcc (Branch Conditionally) instruction with a condition field of 0001.

Example The BSR *+$000A instruction branches to a subroutine at offset $000A beyond the address of the BSR instruction. (The assembly language symbol * refers to the location of the first word of the instruction.) This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	1	8-bit displacement							
0	1	1	0	0	0	0	1	0	0	0	0	1	0	0	0

The instruction has hex value $6108 (the two-byte form). Execution yields:

Before:

```
USP=b03f8a34 MSP=b03f8a34 ISP =b03f8a34 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a34
PC =BSR$        BSR.S      *+$000A      ; BSR$+00a
```

After:

```
USP=b03f8a34 MSP=b03f8a34 ISP =b03f8a30 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a30
PC =BSR$+00a    NOP
```

The change in the stack pointer (A7) accommodates the return address. Since the Status Register specifies the Interrupt Stack Pointer (ISP), the debugger shows this register as also changed .

BTST Instruction

Test a Bit

Syntax: **BTST Dn,<ea>**
BTST #<data>,<ea>

Description

The BTST instruction tests a bit in the destination and sets the Z bit in the condition code register according to the value of the tested bit. The bit number can be specified in a data register or in an immediate field in the instruction itself. The destination can be a data register or an effective address operand. A data register destination implies a long (32-bit) data size. All other destination operands imply a byte (8-bit) data size.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	*	-	-		•		•		•	•	•	•

☐ Data Register Destination only

X Not affected.
N Not affected.
Z Set if the bit tested was zero before the instruction began. Cleared otherwise.
V Not affected.
C Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100	•	
(d16,PC)	111	010		•
(d8,PC,Xn)	111	011		•
(bd,PC,Xn)	111	011		•
([bd,PC,Xn],od)	111	011		•
([bd,PC],Xn,od)	111	011		•

☐ 68020/030 only ☐ Long data size only

Instruction Format

Bit Number Dynamic

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register Dn			1	0	0	Mode			Register		

The Register Dn field contains the number of the data register whose low-order 5 bits (long-data size) or 3 bits (byte-data size) specify the bit number in the destination operand.

Bit Number Static

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	Mode			Register		
0	0	0	0	0	0	0	0	Bit Number							

The bit number is modulo 32. This means that only bits 5–0 (long-data size) or bits 2–0 (byte-data size) of the bit number operand have meaning. All others are ignored.

Example The BTST #$02,D0 instruction tests bit 2 of data register D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	Mode			Register		
0	0	0	0	0	0	0	0	Bit Number							

0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

The instruction has hex value $0800 0002. Execution yields:

Before:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2700 -- -- SI PL=7 -----
D0: 76543210 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =BTST$       BTST       #$02,D0
```

After:

```
USP=b03f8c58 MSP=b03f8c58 ISP =b03f8c58 SR =2704 -- -- SI PL=7 --Z--
D0: 76543210 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8c58
PC =BTST$+004   NOP
```

Since bit 2 of register D0 is zero, the Z condition code bit is set.

CALLM Instruction

Syntax: CALLM #<data>,<ea>

CALL Module

Description

The CALLM instruction allows a fine-grained approach to memory protection. You should be careful in using this instruction since it requires a 68020 equipped with external hardware, such as the 68851 Memory Management Unit described in Chapter 5. Programs using this instruction (and its companion, the RTM (Return from Module) instruction) will be difficult to port to different 68000-based machines.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-								•	

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

The CALLM instruction does not affect the condition codes, has no size attribute, and is supported only by the 68020. Because condition codes are saved and restored across a CALLM / RTM sequence, the called module cannot communicate status using the condition codes.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn			xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100	•	
(An)+	011	An							
-(An)	100	An							
(d16,An)	101	An		•	(d16,PC)	111	010		•
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		•
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		•
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		•
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		•

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	Mode			Register		
0	0	0	0	0	0	0	0	Argument Byte Count							

The effective address operand indicates a module descriptor, and the immediate operand indicates the number of bytes to be passed to the module as arguments.

A module descriptor has the following layout:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$00	CALLM Option			Descriptor Type					Access Level							
$02	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$04	Module Entry Word Pointer (High Word)															
$06	Module Entry Word Pointer (Low Word)															
$08	Module Data Area Pointer (High Word)															
$0A	Module Data Area Pointer (Low Word)															
$0C	Module Stack Pointer (High Word)															
$0E	Module Stack Pointer (Low Word)															
$10	Start of User-Defined Information															

The CALLM option field describes how the CPU is to pass arguments to the called module. A value of 000 indicates that the called module expects to find the arguments below the module stack frame. A value of 100 indicates that the called module expects to access the arguments through an indirect pointer. In cases where a change of stack pointer is required, the 68020 will copy the arguments for format 000, but not for format 100. Any other values for the format field cause a format exception through vector 14.

The descriptor type field also has two possible values, $00 and $01. Any other values cause a format exception. Module descriptor type $00 indicates no change in access permission, and the called module stack frame is built on top of the current stack. Descriptor type $01 indicates a possible stack change and a possible access level change. Descriptor types $10–$1F are reserved for user definition and cause a format error exception. You can thus use the high bit of the descriptor type as a means of disabling a module without disrupting the rest of the module descriptor.

The access level field specifies the desired access level to be used by the called module. This field is passed unchanged and unchecked by the 68020 to external hardware if the descriptor type is $01.

The module entry word contains the address of a descriptor word, which is immediately followed by the first module instruction to be executed. The descriptor word has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	Register			0	0	0	0	0	0	0	0	0	0	0	0
First instruction of called module															

The D/A and Register fields identify a register to receive the module data pointer from the module descriptor area. The D/A bit is zero for a data register and one for an address register. Specifying register A7 effectively defeats this feature because the stack pointer is later overwritten by the value from the module descriptor area. The RTM instruction can also use this technique to avoid modification of any registers when returning from a module.

The CALLM instruction creates an area on the stack known as the Module Call Stack Frame. This area has the following format:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$00	CALLM Option				Descriptor Type				Saved Access Level							
$02	0	0	0	0	0	0	0	0	Saved Condition Code Register							
$04	0	0	0	0	0	0	0	0	Argument Byte Count							
$06	Reserved for expansion															
$08	Module Descriptor Pointer (High Word)															
$0A	Module Descriptor Pointer (Low Word)															
$0C	Saved Program Counter (High Word)															
$0E	Saved Program Counter (Low Word)															
$10	Saved Module Data Area Pointer (High Word)															
$12	Saved Module Data Area Pointer (Low Word)															
$14	Saved Stack Pointer (High Word)															
$16	Saved Stack Pointer (Low Word)															
$18	Arguments (Optional)															

The processor copies the indicated information on a CALLM instruction, and restores this information on the corresponding RTM instruction. Offset zero contains a copy of the CALLM Option and Descriptor type fields from the Module Descriptor. The Access Level contains the access level before the CALLM instruction. Offset 2 contains a copy of the Condition Code Register before the CALLM instruction. Offset 4 contains a copy of the argument byte count from the second word of the CALLM instruction. Offset 6 is reserved for future expansion.

Offsets 8–$0B contain the address of the module descriptor. Offsets $0C–$0F contain the saved program counter, which contains the return address. Offsets $10–$13 contain the original contents of the register specified in the module entry word shown above. This longword is restored into a register designated by the RTM instruction. Offsets $14–$17 contain the original contents of the stack pointer before the CALLM instruction. This quantity is restored into A7 by the RTM instruction.

Descriptor Type $00 Calling Sequence If the Descriptor Type field in the Module Descriptor area is $00, no stack switch or access level change is performed. Regardless of the setting of the CALLM Option field in the Module Descriptor, no arguments are copied, and the Module Stack Frame is built using the current value of A7.

Descriptor Type $01 Calling Sequence For a Module Descriptor area with a type field of $01, there is a possibility of both an access level change and/or a stack switch. When processing this type of descriptor, the 68020 communicates with external hardware using CPU space bus cycles with the address map shown below.

	31 24	23 0
$00	CAL	(Unused, Reserved)
$04	STATUS	(Unused, Reserved)
$08	IAL	(Unused, Reserved)
$0C	DAL	(Unused, Reserved)

$40	Function Code 0 Descriptor Address
$44	Function Code 1 Descriptor Address (User Data)
$48	Function Code 2 Descriptor Address (User Program)
$4C	Function Code 3 Descriptor Address
$50	Function Code 4 Descriptor Address (Supervisor Data)
$54	Function Code 5 Descriptor Address (Supervisor Program)
$58	Function Code 6 Descriptor Address
$5C	Function Code 7 Descriptor Address (CPU Space)

The Current Access Level (CAL) contains the access rights (an 8-bit number) of the current module. The Increase Access Level (IAL) and Decrease Access Level (DAL) registers allow the processor to request higher and lower access levels, respectively. The processor simply passes the access level field from the module descriptor without interpretation to IAL or DAL, as appropriate.

The Status field (offset 4) gives the processor the status of an access change request, as follows:

Value	Valid?	Resulting Processor Action
00	No	Take format error exception
01	Yes	No change in access rights
02-03	Yes	Change access rights without changing stack pointer
04-07	Yes	Change access rights and stack pointer
Other	Undefined	Take format error exception

The processor uses the descriptor addresses to communicate the address of the type $01 descriptor. This prevents the calling module from using a counterfeit descriptor to gain a higher access level illegally.

Example The CALLM #8,*+$0010 uses the descriptor located at the instruction address plus $10 to call a module. (The assembly language symbol * refers to the location of the first word of the instruction.) This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	Mode			Register		
0	0	0	0	0	0	0	0	Argument Byte Count							

0	0	0	0	0	1	1	0	1	1	1	1	1	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0

The instruction has hex value $06FA 0008 000C. Note that the offset from the PC is contained in the last word and that this offset is relative to the second word of the instruction. Execution yields:

Before:

```
USP=b03f85ac MSP=b03f85ac ISP =b03f85a4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f85a4
PC =CALLM$      CALLM       #$08,*+$0010         ; CALLM$DES
```

This instruction passes 8 bytes of arguments to the module, located on the stack as follows:

```
b03f85a4:          fedcba98 76543210
```

The module descriptor has the following format:

```
CALLM$DES:         80000000 b03f8a16 a0a0a0a0 00000000 .....?..........
```

This example uses the argument copy option (which is ignored) and a descriptor type of zero. Register A0 serves as the module data pointer and receives a new value of $a0a0a0a0 (offset 8). The descriptor (and the called module) are located at address b03f8a16 (offset 4), as shown below:

```
CALLM$EE:          8000 4e71
```

The first word of the descriptor specifies register A0. The next word is the first instruction (a NOP).

After:

```
USP=b03f85ac MSP=b03f85ac ISP =b03f858c SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: a0a0a0a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f858c
PC =CALLM$EE+002          NOP
```

Note that the stack has grown by $18 bytes. The top of the stack is as follows:

```
b03f858c:          8000 0000 0008 00a5 b03f 89ee b03f 89e4
b03f859c:          0000 00a0 b03f 85a4 fedc ba98 7654 3210
```

The top of the stack contains a copy of the option word from the module descriptor ($8000). The next word contains the original contents of the Condition Code Register ($0000). Offset 6 into the stack is undefined (in this case, $00A5). Offsets $8–$B ($B03F89EE) contain the address of the module descriptor (CALLM$DES as shown above). Offsets $C–$F ($B03F89E4) are the return address (CALLM$+6). Offsets $11–$13 contain the original contents of A0. The original contents of the stack pointer occupy the next longword (Offsets $14–$17). Note that the arguments are at offset $18.

CAS Instruction

Syntax: CAS Dc,Du,<ea>

Compare and Swap (One Operand)

Description

The CAS instruction allows a programmer to update a single operand in an atomic fashion in a multiprocessor system. The instruction works like this: The effective address is compared to a data register (the comparison value, Dc). If the two are equal, the operand is replaced by a second data register (the update value, Du). If the two are not equal, the comparison register, Dc, is set to the value located at the effective address. The fetch, comparison, and update are all performed without releasing the memory bus to other processors. This is termed a *read/modify/write* bus cycle and guarantees that no processor can update the operand until the instruction has completed. All processors modifying a given memory location must use read/modify/write bus cycles to ensure proper synchronization.

The condition codes are set according to the results of the comparison. If the Z bit is not set, the values in the comparison were not equal and the update did not take place. The issuing processor may then take appropriate action for synchronization failure.

Condition Codes	X	N	Z	V	C	**Sizes**	Byte	Word	Long	**CPUs**	000	010	020	030
	-	*	*	*	*		•	•	•				•	•

X	Not affected.
N	Set if result of comparison is negative (bit 31/15/7 = 1). Cleared otherwise.
Z	Set if result of comparison is zero. Cleared otherwise.
V	Set if comparison generated an overflow. Cleared otherwise.
C	Set if comparison generated an overflow. Cleared otherwise.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•		xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100		
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="2">Size</td><td>0</td><td>1</td><td>1</td><td colspan="3">Mode</td><td colspan="3">Register</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="3">Du Register</td><td>0</td><td>0</td><td>0</td><td colspan="3">Dc Register</td></tr>
</table>

The Size field (bits 10–9) is 01 for byte data size, 10 for word, and 11 for long.

Example The CAS.W D1,D0,(A0) instruction compares the word at the address in A0 to D1.W. If the comparison is equal, the memory operand is replaced by the contents of D0.W. This instruction has the following layout:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="2">Size</td><td>0</td><td>1</td><td>1</td><td colspan="3">Mode</td><td colspan="3">Register</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="3">Du Register</td><td>0</td><td>0</td><td>0</td><td colspan="3">Dc Register</td></tr>
</table>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	1	1	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

The instruction has hex value $0CD0 0001. Execution yields:

Before:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2700 -- -- SI PL=7 -----
D0: 00000001 00000000 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8ef4 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CAS$        CAS.W      D1,D0,(A0)
```

The memory operand has the following value:

```
b03f8ef4:       0000
```

After:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2704 -- -- SI PL=7 --Z--
D0: 00000001 00000000 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8ef4 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CAS$+004    NOP
```

The memory operand now has the following value:

```
b03f8ef4:       0001
```

CAS2 Instruction Syntax: CAS2 Dc1:Dc2,Du1:Du2,(Rn1):(Rn2)

Compare and Swap (Two Operands)

Description

The CAS2 instruction allows a programmer to update two operands in an atomic fashion in a multiprocessor system. The instruction works like this: There are six registers involved, two comparison values, two update values, and two addresses. The two comparison values are compared against the contents of the two memory addresses. If both comparisons succeed, the contents of the memory locations are replaced by the two update values. If either comparison fails, the two comparison operands are replaced by the contents of the memory locations. No other processors may gain access to the two memory locations until the instruction has completed. This ensures proper synchronization in a multiprocessor system.

The condition codes are set according to the results of the comparisons. If the Z bit is not set, the values in the comparisons were not equal and the memory update did not take place. The issuing processor may then take appropriate action for synchronization failure.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	*			•	•				•	•

X Not affected.
N Set if result of comparison is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result of comparison is zero. Cleared otherwise.
V Set if comparison generated an overflow. Cleared otherwise.
C Set if comparison generated an overflow. Cleared otherwise.

This instruction has no effective address field. The operands are constrained to reside in registers, as described above.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	Size		0	1	1	1	1	1	1	0	0
D/A1	Register Rn1			0	0	0	Du Register			0	0	0	Dc Register		
D/A2	Register Rn2			0	0	0	Du Register			0	0	0	Dc Register		

The Size field (bits 10–9) is 10 for word data and 11 for long. The D/A1 and D/A2 bits indicate whether register Rn1 or Rn2 (respectively) is a data or address register. These bits are zero for a data register and one to indicate an address register.

Example The CAS2.L D2:D3,D0:D1,(A0):(A1) instruction compares the contents of register D2 with the memory location whose address is in A0 and the contents of register D3 with the memory location whose address is in A1. If both comparisons are equal, the two memory locations are replaced by the contents of registers D0 and D1. If either comparison is not equal, D2 and D3 are set to the contents of the memory locations. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	Size		0	1	1	1	1	1	1	0	0
D/A1	Register Rn1			0	0	0	Du Register			0	0	0	Dc Register		
D/A2	Register Rn2			0	0	0	Du Register			0	0	0	Dc Register		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	1	1	1	1	1	1	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	1	0	0	0	0	1	1

The instruction has hex value $0EFC 8002 9043. Execution yields:

Before:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2700 -- -- SI PL=7 -----
D0: 00000000 00000001 00000002 00000003 000000d4 000000d5 000000d6 000000d7
A0: b03f8ef4 b03f8ef8 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CAS2$     CAS2.L   D2:D3,D0:D1,(A0):(A1)
```

The memory locations contain 0 and 1, as shown below:

```
b03f8ef4:     00000000
b03f8ef8:     00000001
```

After:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2709 -- -- SI PL=7 -N--C
D0: 00000000 00000001 00000000 00000001 000000d4 000000d5 000000d6 000000d7
A0: b03f8ef4 b03f8ef8 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CAS2$+006          NOP
```

Since the registers were not equal to the memory locations, the register contents are replaced by the contents of the memory locations. The memory locations are unchanged.

CHK Instruction

Syntax: CHK <ea>,Dn

Check Register Against Bounds (One Operand)

Description

The CHK instruction tests a data register against a positive range. If the data register is less than zero or greater than the value of the effective address operand, the processor takes a CHK exception through vector 6.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	U	U	U			•	•		•	•	•	•

X	Not affected.
N	Set if Dn < 0. Cleared if Dn > <ea>. Undefined otherwise.
Z	Undefined.
V	Undefined.
C	Undefined.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An	•	
(An)+	011	An	•	
-(An)	100	An	•	
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	110	An	•	
([bd,An],Xn,od)	110	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100	•	
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	Register			Size		0	Mode			Register		

11=Word
10=Long

The data register identifier resides in bits 11–9. Bits 8–7 give the operation size (11 for word, 10 for long).

Example The CHK.L #$2,D0 instruction causes a CHK exception if the contents of D0 are less than zero or greater than two. The instruction has the following layout:

Instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	Register			Size		0	Mode			Register		
0	1	0	0	0	0	0	1	0	0	1	1	1	1	0	0

Data

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

The hex value of the instruction is $413C 0000 0002. Execution yields:

Before:

```
USP=b03f8a9a MSP=b03f8a9a ISP =b03f8a9a SR =2700 -- -- SI PL=7 -----
D0: 00000003 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a9a
PC =CHK$     CHK.L  #$00000002,D0
```

After:

```
CHK Instruction Failure
USP=b03f8a9a MSP=b03f8a9a ISP =b03f8a9a SR =a700 T1 -- SI PL=7 -----
D0: 00000003 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a9a
PC =CHK$+006 NOP
```

Since D0 has the value 3, which is out of range, the instruction caused an exception through vector 6. The stacked value of the PC points to the next instruction.

CHK2 Instruction

Syntax: CHK2 <ea>,Rn

Check Register Against Bounds (Two Operands)

Description

The CHK2 instruction tests a register against a 32-bit range. If the data register is either less than the lower bound or greater than the value of the upper bound, the processor takes a CHK exception through vector 6. The register can be either an address register or data register. The effective address operand identifies a pair of longwords in memory. The base value occurs at the effective address and the bounds value is located at the effective address plus 4.

CHK2 can be used for both signed and unsigned comparisons. For signed comparisons, use the arithmetically (two's complement) smaller value as the lower bound. For unsigned comparisons, use the (unsigned) logically smaller value as the lower bound.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	U	*	U	*		•	•	•				•	•

X Not affected.
N Undefined.
Z Set if Rn equal to either bound. Cleared otherwise.
V Undefined.
C Set if Rn out of bounds. Cleared otherwise.

When an address register is specified as the destination, and the size is byte or word, the bounds values will be sign-extended to 32 bits and compared with all 32 bits of the register.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000	•	
An	001	An		•	xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100		
(An)+	011	An							
-(An)	100	An							
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	Size		0	1	1	Mode			Register		
D/A	Register			1	0	0	0	0	0	0	0	0	0	0	0

Bits 10–9 give the operation size (00 for byte, 01 for word, and 10 for long). The D/A and Register fields in the extension word specify the register to be checked. The D/A bit is zero for a data register and one for an address register.

Example The CHK2.L (A0),D0 instruction causes a CHK exception if the contents of D0 are outside the range whose address is in A0. The instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	Size		0	1	1	Mode			Register		
D/A	Register			1	0	0	0	0	0	0	0	0	0	0	0

0	0	0	0	0	1	0	0	1	1	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

The hex value of the instruction is $04D0 0800. Execution yields:

Before:

```
USP=b03f8a9a MSP=b03f8a9a ISP =b03f8a9a SR =2700 -- -- SI PL=7 -----
D0: 00000000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8f3c 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a9a
PC =CHK2$      CHK2.L   (A0),D0
```

The memory locations contain 0 and 1, as shown below.

```
b03f8f3c:      00000000
b03f8f40:      00000001
```

After:

```
USP=b03f8a9a MSP=b03f8a9a ISP =b03f8a9a SR =2704 -- -- SI PL=7 --Z--
D0: 00000000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8f3c 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a9a
PC =CHK2$+004  NOP
```

Since D0 has the value 0, which is equal to one of the bounds, the Z condition code bit was set.

CLR Instruction

Clear (Zero) an Operand

Syntax: CLR <ea>

Description

The CLR instruction sets an effective address operand to zero. Some 68000 family chips perform a read cycle before the write cycle on memory operands. This may require a MOVE #0 instruction instead of a CLR on some types of hardware.

The MPW assembler substitutes a MOVEQ #0,Dn instruction for a CLR Dn instruction.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	0	1	0	0		•	•	•		•	•	•	•

X Not affected.
N Always cleared.
Z Always set.
V Always cleared.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100		
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

The CLR instruction has no source operand.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	Size		Mode			Register		

00=Byte
01=Word
10=Long

Example The CLR.L D0 instruction clears all bits of register D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	Size		Mode			Register		
0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0

The instruction has hex value $4280. Execution yields:

Before:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =CLR$        CLR.L D0
```

After:

```
USP=b03f8b72 MSP=b03f8b72 ISP =b03f8b72 SR =2704 -- -- SI PL=7 --Z--
D0: 00000000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8b72
PC =CLR$+002    NOP
```

CMP Instruction

Compare with Data Register

Syntax: CMP <ea>,Dn

Description

The CMP instruction sets the condition codes according to the relationship between the contents of a data register and the contents of the effective address operand. The effective address operand is subtracted from the data register, and the condition codes are set accordingly. Neither operand is changed by the instruction.

The 68000 family compare and conditional branch instructions are somewhat difficult to read. The instruction sequence

```
CMP.L    D0,D1
BGT      SOMEWHERE
```

branches to SOMEWHERE if D1 is greater than D0. This is contrary to the way you read most of the other 68000 instructions. Some assemblers (notably A/UX) attempt to rectify this situation by inverting the order of the operands. The compare instruction above would be written CMP.L D1,D0 to generate the same machine code on these assemblers. This approach makes the code more readable, but is incompatible with other systems.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	*		•	•	•		•	•	•	•

X Not affected.
N Set if the result is negative (bit 31/15/7 = 1) Cleared otherwise.
Z Set if the operands are equal. Cleared otherwise.
V Set if an overflow was generated. Cleared otherwise.
C Set if a borrow was generated. Cleared otherwise.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An *	001	An	•	
(An)	010	An	•	
(An)+	011	An	•	
-(An)	100	An	•	
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	**110**	An	•	
([bd,An],Xn,od)	**110**	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100	•	
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	**111**	**011**	•	
([bd,PC],Xn,od)	**111**	**011**	•	

* Word or long only ☐ 68020/030 only

When you specify a CMP instruction with an address register as a destination, many assemblers will automatically generate the CMPA instruction.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	Data Register			0	Size		Mode			Register		

00=Byte
01=Word
10=Long

Example The CMP.L D0,D1 instruction compares register D0 to register D1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	Data Register			0	Size		Mode			Register		
1	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0

The hex value of the instruction is $B280. Executing this instruction yields:

Before:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2700 -- -- SI PL=7 -----
D0: 00000001 00000000 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CMP$        CMP.L      D0,D1
```

After:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2709 -- -- SI PL=7 -N--C
D0: 00000001 00000000 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CMP$+002    NOP
```

Note the condition code settings. The N and C bits indicate that register D1 is less than register D0.

CMPA Instruction

Compare with Address Register

Syntax: CMPA <ea>,An

Description

The CMPA instruction sets the condition codes according to the relationship between the contents of an address register and the contents of the effective address operand. The effective address operand is subtracted from the address register, and the condition codes are set accordingly. Neither operand is changed by the instruction.

The 68000 family compare and conditional branch instructions are somewhat difficult to read. The instruction sequence

```
CMPA.L          D0,A0
BGT             SOMEWHERE
```

branches to SOMEWHERE if A0 is greater than D0. This is contrary to the way you read most of the other 68000 instructions. Some assemblers (notably A/UX) attempt to rectify this situation by inverting the order of the operands. The compare instruction above would be written CMPA.L A0,D0 to generate the same machine code on these assemblers. This approach makes the code more readable, but is incompatible with other systems.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	*			•	•		•	•	•	•

X	Not affected.
N	Set if the result is negative (bit 31 = 1) Cleared otherwise.
Z	Set if the operands are equal. Cleared otherwise.
V	Set if an overflow was generated. Cleared otherwise.
C	Set if a borrow was generated. Cleared otherwise.

Word operands are sign-extended to 32 bits, and the comparison takes place with all 32 bits of the address register.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	
An	001	An	•	•
(An)	010	An	•	
(An)+	011	An	•	
-(An)	100	An	•	
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	110	An	•	
([bd,An],Xn,od)	110	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100	•	
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	Address Reg.			Size	1	1	Mode			Register		

0 = Word
1 = Long

Example The CMPA.L D0,A0 instruction compares register D0 to register A0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	Address Reg.			1	1	1	Mode			Register		
1	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0

The hex value of the instruction is $B1C0. Executing the instruction yields:

Before:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2700 -- -- SI PL=7 -----
D0: 00000001 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 00000000 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CMPA$     CMPA.L   D0,A0
```

After:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2709 -- -- SI PL=7 -N--C
D0: 00000001 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 00000000 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CMPA$+002       NOP
```

Note the condition code settings. The N and C bits indicate that register A0 is less than register D0.

CMPI Instruction

Compare Immediate

Syntax: CMPI #<data>,<ea>

Description

The CMPI instruction compares immediate data with an effective address operand. The immediate operand is subtracted from the effective address operand, and the condition codes are set accordingly. Neither operand is changed by the instruction.

The 68000 family compare and conditional branch instructions are somewhat difficult to read. The instruction sequence

```
CMPI.L   #data,D0
BGT      SOMEWHERE
```

branches to SOMEWHERE if D0 is greater than the immediate quantity. This is contrary to the way you read most of the other 68000 instructions. Some assemblers (notably A/UX) attempt to rectify this situation by inverting the order of the operands. The compare instruction above would be written CMPI.L D0,#data to generate the same machine code on these assemblers. This approach makes the code more readable, but is incompatible with other systems.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	*		●	●	●		●	○	○	○

X	Not affected.
N	Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z	Set if the operands are equal. Cleared otherwise.
V	Set if an overflow was generated. Cleared otherwise.
C	Set if a carry was generated. Cleared otherwise.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100	•	
(An)+	011	An		•					
-(An)	100	An		•					•
(d16,An)	101	An		•	(d16,PC)	111	010		•
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		•
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		•
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		•
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		•

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	Size		Mode			Register		
Word data (16 bits)								Byte data (8 bits)							
Long data															

The size field indicates the operation size: 00 for byte, 01 for word, and 10 for long. There are 2 or 4 bytes of immediate data that follow the instruction in memory. Byte and word operations have 16 bits (2 bytes) of immediate data. Long operations have 32 bits (4 bytes).

Example The instruction CMPI.L #$1,D0 compares the hex quantity $1 with all 32 bits of register D0. The immediate data ($1) is contained in the two words following the instruction, as shown below:

Instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	Size		Mode			Register		
0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0

Data

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

This instruction has hex value $0C80 0000 0001. Executing the instruction yields:

Before:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2700 -- -- SI PL=7 -----
D0: 00000000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CMPI$       CMPI.L     #$00000001,D0
```

After:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2709 -- -- SI PL=7 -N--C
D0: 00000000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CMPI$+006   NOP
```

CMPM Instruction

Syntax: CMPM (Ay)+,(Ax)+

Compare Memory

Description

The CMPM instruction compares two memory locations specified by the contents of two address registers. This instruction is used for multiple-precision compare operations. The longword whose address is in Ay is subtracted from the longword whose address is in Ax. The condition codes are set according to the results of this subtraction.

The 68000 big-endian data format dictates the form of multiple-precision comparisons with the CMPM instruction. The comparison must begin at the lowest-addressed byte and work forward to the highest-addressed byte, using postincrement addressing.

The 68000 family compare and conditional branch instructions are somewhat difficult to read. The instruction sequence

```
CMPM.L   (A0)+,(A1)+
BGT      SOMEWHERE
```

branches to SOMEWHERE if A1 points to a longword greater than the longword whose address is in A0. This is contrary to the way you read most of the other 68000 instructions. Some assemblers (notably A/UX) attempt to rectify this situation by inverting the order of the operands. The compare instruction above would be written CMPM.L (A1)+,(A0)+ to generate the same machine code on these assemblers. This approach makes the code more readable, but is incompatible with other systems.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	*		•	•	•		•	•	•	•

X Not affected.
N Set if result is negative (bit 31 = 1). Cleared otherwise.
Z Set if result was zero. Cleared otherwise.
V Set if an overflow was generated. Cleared otherwise.
C Set if a carry was generated. Cleared otherwise.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	Register Ax			1	Size		0	0	1	RegisterAy		

00=Byte
01=Word
10=Long

Bits 11–9 contain the destination register number, and bits 2–0 contain the source register number. The condition codes are set by subtracting the source from the destination.

Example The CMPM.L (A0)+,(A1)+ compares the longwords addressed by A0 and A1. Both address registers are incremented by 4 following the instruction, and the condition codes are set as described above.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	Register Ax			1	Size		0	0	1	Register Ay		
1	0	1	1	0	0	1	1	1	0	0	0	1	0	0	0

The CMPM instruction has hex value $B388. Executing the instruction yields:

Before:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8f66 b03f8f6a 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CMPM$      CMPM.L     (A0)+,(A1)+
```

Note that the two address registers point to consecutive longwords. The contents of these memory locations are shown below. The instruction compares a longword containing zero to a longword containing one.

```
b03f8f66:     00000000 00000001
```

After:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8f6a b03f8f6e 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CMPM$+002   NOP
```

No condition code bits are set as a result of this comparison. A BGT (Branch Greater Than) or a BHI (Branch Higher) instruction will branch following such a comparison, indicating that the destination operand was larger than the source.

CMP2 Instruction

Syntax: CMP2 <ea>,Rn

Compare Register Against Bounds

Description

The CMP2 instruction tests a register against a 32-bit range. The condition codes are set according to the comparison. The effective address operand identifies a pair of longwords in memory. The base value occurs at the effective address and the bounds value is located at the effective address plus 4.

CMP2 can be used for both signed and unsigned comparisons. For signed comparisons, use the arithmetically (two's complement) smaller value as the lower bound. For unsigned comparisons, use the (unsigned) logically smaller value as the lower bound.

This instruction is essentially the same as a CHK2 instruction, except that the C (Carry) condition code bit is set if the register is outside the range, rather than having the processor take an exception.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	U	*	U	*		•	•	•				•	•

X	Not affected.
N	Undefined.
Z	Set if Rn equal to either bound. Cleared otherwise.
V	Undefined.
C	Set if Rn out of bounds. Cleared otherwise.

When an address register is specified as the destination, and the size is byte or word, the bounds values will be sign extended to 32 bits and compared with all 32 bits of the register.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000	•	
An	001	An		•	xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100		
(An)+	011	An							
-(An)	100	An							
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	Size		0	1	1	Mode			Register		
D/A	Register			0	0	0	0	0	0	0	0	0	0	0	0

Bits 10–9 give the operation size (00 for byte, 01 for word, and 10 for long). The D/A and Register fields in the extension word specify the register to be checked. The D/A bit is zero for a data register and one for an address register.

Example The CMP2.L (A0),D0 compares data register D0 to two longwords addressed by A0. The condition codes are set as described above.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	Size		0	1	1	Mode			Register		
D/A	Register			0	0	0	0	0	0	0	0	0	0	0	0

0	0	0	0	0	1	0	0	1	1	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The instruction has hex value $04D0 0000. Execution yields:

Before:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2700 -- -- SI PL=7 -----
D0: 00000000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8f66 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CMP2$       CMP2.L     (A0),D0
```

A0 points to two consecutive longwords. The contents of these memory locations are shown below. The instruction compares D0 to a longword containing zero and a longword containing one.

```
b03f8f66:     00000000 00000001
```

After:

```
USP=b03f8ae6 MSP=b03f8ae6 ISP =b03f8ae6 SR =2704 -- -- SI PL=7 --Z--
D0: 00000000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8f66 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8ae6
PC =CMP2$+004   NOP
```

Since D0 is equal to one of the bounds, the Z condition code bit is set by the instruction.

cpBcc Instruction

Branch on Coprocessor Condition

Syntax: cpBcc <label>

Description

The cpBcc instruction branches if a coprocessor condition is true. This instruction is similar to the Bcc instructions, except that the condition is defined by the coprocessor and there is no byte displacement form of the instruction.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-			•	•				•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Coprocessor ID			0	1	Size	Coprocessor Condition					
Optional coprocessor extension words															
16-bit displacement if size bit = 0															
32-bit displacement if size bit = 1															

The branch instructions conditionally alter the value of the program counter (PC) register depending on the setting of the coprocessor condition codes. If the coprocessor indicates the branch should be taken, the displacement is added to the PC to yield the address of the next instruction to be executed. The PC always points to the first word of the displacement for this calculation.

Example Refer to the FBcc instruction in Chapter 4.

cpDBcc Instruction

Syntax: cpDBcc Dn,<label>

Test Coprocessor Condition, Decrement, and Branch

Description

The cpDBcc instruction evaluates whether a coprocessor condition is true. If true, execution proceeds to the next instruction. If not true, the cpDBcc instruction decrements the low word of the specified data register and branches if this word does not contain -1 following the decrement. This instruction is similar to the DBcc instructions, except that the condition is defined by the coprocessor.

A cpDBcc instruction is intended for use at the end of a program loop. The loop can terminate on one of two conditions: the number of iterations in the data register, or the coprocessor condition.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-			•					•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Coprocessor ID			0	0	1	0	0	1	Register		
0	0	0	0	0	0	0	0	0	0	Coprocessor Condition					
Optional coprocessor extension words															
16-bit displacement															

The cpDBcc instructions conditionally alter the value of the program counter (PC) register depending on the setting of the coprocessor condition codes or the count register. If the coprocessor indicates the condition is false, the sign-extended displacement is added to the PC to yield the address of the next instruction to be executed. The PC always points to the displacement word for this calculation.

Example Refer to the FDBcc instruction in Chapter 4.

cpGEN Instruction

Syntax: cpGEN <parameters>

Pass Command Word to Coprocessor

Description

The cpGEN instruction provides coprocessors with the ability to extend the 68000 instruction set in a general-purpose way. This instruction provides a 16-bit command word for the coprocessor and also allows coprocessor instruction execution in parallel with main processor execution. Slow coprocessor instructions can be overlapped with standard 68000 instructions.

<table>
<tr><th rowspan="2">Condition Codes</th><th>X</th><th>N</th><th>Z</th><th>V</th><th>C</th><th rowspan="2">Sizes</th><th>Byte</th><th>Word</th><th>Long</th><th rowspan="2">CPUs</th><th>000</th><th>010</th><th>020</th><th>030</th></tr>
<tr><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td></td><td></td><td></td><td></td><td></td><td>•</td><td>•</td></tr>
</table>

X Not affected unless modified by coprocessor.
N Not affected unless modified by coprocessor.
Z Not affected unless modified by coprocessor.
V Not affected unless modified by coprocessor.
C Not affected unless modified by coprocessor.

The cpGEN instruction has no size attribute.

Instruction Format

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="3">Coprocessor ID</td><td>0</td><td>0</td><td>0</td><td colspan="3">Mode</td><td colspan="3">Register</td></tr>
<tr><td colspan="16">Coprocessor-dependent command word</td></tr>
<tr><td colspan="16">Optional effective address or coprocessor-defined extension words</td></tr>
</table>

The effective address field specifies the location of an operand that does not reside in the coprocessor. Allowable addressing modes depend on the operation to be performed. The coprocessor command word is passed to the coprocessor, which then communicates with the main processor while executing the instruction.

Example Almost all of the floating point instructions in Chapter 4 are cpGEN instructions. The exceptions are FBcc, FDBcc, FSAVE, FRESTORE, FScc, and FTRAPcc.

cpRESTORE Instruction (Privileged) Syntax: cpRESTORE <ea>

Restore Coprocessor State

Description

The cpRESTORE instruction resets the coprocessor to the state preserved by a previous cpSAVE instruction. The processor must be in Supervisor mode to execute this instruction. Attempted execution in user mode will cause a Privilege Violation exception through vector 8.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-								•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

The cpRESTORE instruction has no size attribute.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn			xxx.W	111	000	•	
An	001	An			xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100		
(An)+	011	An	•						
-(An)	100	An							
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Coprocessor ID			1	0	1	Mode			Register		

Example Refer to the FRESTORE instruction in Chapter 4.

cpSAVE Instruction (Privileged) Syntax: cpSAVE <ea>

Save Coprocessor State

Description

The cpSAVE instruction preserves the coprocessor state in a form suitable for restoration by a cpRESTORE instruction. The processor must be in Supervisor mode to execute this instruction. Attempted execution in user mode will cause a Privilege Violation exception through vector 8.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-								•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

The cpSAVE instruction has no size attribute.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn			xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100		
(An)+	011	An							
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="3">Coprocessor ID</td><td>1</td><td>0</td><td>0</td><td colspan="3">Mode</td><td colspan="3">Register</td></tr>
</table>

Example Refer to the FSAVE instruction in Chapter 4.

cpScc Instruction

Set on Coprocessor Condition

Syntax: cpScc <ea>

Description

The cpScc instruction sets a byte to all zeros or all ones, depending on a coprocessor condition. The byte is set to zero if the coprocessor evaluates the condition as false and to all ones if the condition is true. This instruction is similar to the Scc instructions, except that the condition is evaluated by the coprocessor.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-								•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100		
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Coprocessor ID			0	0	1	Mode			Register		
0	0	0	0	0	0	0	0	0	0	Coprocessor Condition					
Optional coprocessor extension words															

Example Refer to the FScc instruction in Chapter 4.

cpTRAPcc Instruction

TRAP on Coprocessor Condition

Syntax: cpTRAPcc #<data>
cpTRAPcc

Description

The cpTRAPcc instruction tests a coprocessor condition. If the coprocessor indicates the condition is true, the main processor takes a cpTRAPcc exception through vector 7. If the condition is false, the main processor begins execution of the next instruction. This instruction is similar to the TRAPcc instructions, except that the condition is evaluated by the coprocessor.

The cpTRAPcc exception uses a format 2 stack frame, as shown in Figure 2–39. The program counter value placed on the stack is the address of the next instruction.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-			•	•				•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

A cpTRAPcc instruction can be followed by zero, 2, or 4 bytes of immediate data for use in processing the exception.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Coprocessor ID			0	0	1	1	1	1	Op–Mode		
0	0	0	0	0	0	0	0	0	0	Coprocessor Condition					
Optional coprocessor extension words															
Optional 16-bit															
or 32-bit immediate data															

The Op-Mode field specifies the presence or absence of immediate data. This field is 100 if no immediate data is present, 010 for 16 bits of immediate data, and 011 for 32 bits of immediate data.

Example Refer to the FTRAPcc instruction in Chapter 4.

DBcc Instruction

Test Condition, Decrement, and Branch

Syntax: DBcc Dn,<label>

Description

The DBcc instructions test a main processor condition. If the specified condition is false, then the low word of the data register is decremented. If the resulting value is not -1, a branch to the specified label is taken. DBcc is normally used as the last instruction in a loop. Note that an initial value of -1 in the data register will cause 65,536 iterations (the largest possible value using DBcc) of the loop.

The condition may be viewed as the termination condition of a loop, much like the do/while construct in C, or the REPEAT/UNTIL construct in the Pascal language. Unlike the language constructs, the DBcc condition terminates the loop when the condition is true rather than when false.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-			•			•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	Condition				1	1	0	0	1	Data Register		
16-bit displacement															

The DBcc instructions conditionally change the value of the program counter (PC) register depending on the setting of the condition codes and the value in the data register. The value used for the PC is the address of the second word of the instruction. If the branch is taken, the sign-extended value of the 16-bit displacement is added to the PC to obtain the address of the next instruction.

Conditions The condition field (bits 11–8) indicates the combination of condition codes desired. Four bits allow sixteen possible combinations. Unlike the Bcc instructions, all sixteen combinations are possible. The conditions and the corresponding instructions in the DBcc family are listed below.

Condition	**Meaning**
0000 ($0)	**DBT** (**D**ecrement and **B**ranch **T**rue). This instruction never branches because the termination condition is always true.
0001 ($1)	**DBF** (**D**ecrement and **B**ranch **F**alse). Because the termination condition for this instruction is always false, the loop terminates only on the count in the data register. Many assemblers also accept the **DBRA** mnemonic for this form of the instruction.
0010 ($2)	**DBHI** (**D**ecrement and **B**ranch **HI**gher). The loop terminates if both the Carry and Zero condition code bits are clear. DBHI is equivalent to a **D**ecrement and **B**ranch **G**reater **T**han (**DBGT**) for unsigned numbers.
0011 ($3)	**DBLS** (**D**ecrement and **B**ranch **L**ower or **S**ame). The loop terminates if either the Carry or Zero condition code bits are set. DBLS is equivalent to **D**ecrement and **B**ranch **L**ess than or **E**qual (**DBLE**) for unsigned numbers.
0100 ($4)	**DBCC** (**D**ecrement and **B**ranch if **C**arry **C**lear). The loop terminates if the Carry bit is not set.
0101 ($5)	**DBCS** (**D**ecrement and **B**ranch if **C**arry **S**et). The loop terminates if the Carry bit is set.
0110 ($6)	**DBNE** (**D**ecrement and **B**ranch if **N**ot **E**qual). The loop terminates if the Z bit is not set.
0111 ($7)	**DBEQ** (**D**ecrement and **B**ranch if **EQ**ual). The loop terminates if the Z bit is set.
1000 ($8)	**DBVC** (**D**ecrement and **B**ranch if o**V**erflow **C**lear). The loop terminates if the V bit is not set.
1001 ($9)	**DBVS** (**D**ecrement and **B**ranch if o**V**erflow **S**et). The loop terminates if the V bit is set.
1010 ($A)	**DBPL** (**D**ecrement and **B**ranch if **PL**us). The loop terminates if the N bit is not set.
1011 ($B)	**DBMI** (**D**ecrement and **B**ranch if **MI**nus). The loop terminates if the N bit is set.
1100 ($C)	**DBGE** (**D**ecrement and **B**ranch if **G**reater than or **E**qual). The loop terminates if the N and V bits are both set, or if the N and V bits are both clear.

1101 ($D)	**DBLT** (**D**ecrement and **B**ranch if **L**ess **T**han). The loop terminates if the N bit is set and the V bit is clear or if the N bit is clear and the V bit is set.
1110 ($E)	**DBGT** (**D**ecrement and **B**ranch if **G**reater **T**han). The loop terminates if the N and V bits are both set and the Z bit is clear or if the N, V, and Z bits are all clear.
1111 ($F)	**DBLE** (**D**ecrement and **B**ranch if **L**ess than or **E**qual). The loop terminates if the Z bit is set or if the N bit is set and the V bit is clear, or if the N bit is clear and the V bit is set.

Example The DBF D0,*-$0002 terminates a loop when D0 is decremented to -1. (The assembly language symbol * refers to the location of the first word of the instruction.) This instruction has the following layout:

0	1	0	1	Condition	1	1	0	0	1	Data Register
16-bit displacement										

0	1	0	1	0	0	0	1	1	1	0	0	1	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0

The instruction has hex value $51C8 FFFC. Execution yields:

Before:

```
USP=b03f8a9a MSP=b03f8a9a ISP =b03f8a9a SR =2700 -- -- SI PL=7 -----
D0: 12340000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a9a
PC =DBRA$       DBF        D0,*-$0002    ; DB_TARGET
```

After:

```
USP=b03f8a9a MSP=b03f8a9a ISP =b03f8a9a SR =2700 -- -- SI PL=7 -----
D0: 1234ffff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a9a
PC =DBRA$+004   NOP
```

Since D0 was zero at entry, the branch is not taken. Note that only the low 16 bits of D0 were decremented.

DIVS Instruction

Signed Divide (Word Form)

Syntax: DIVS.W <ea>,Dn

Description

The DIVS.W instruction divides a 32-bit quantity by a 16-bit quantity, yielding a 16-bit quotient and a 16-bit remainder. The effective address operand is always the divisor, and the data register is always the dividend. The quotient (dividend/divisor) is placed in the low word of the data register, and the remainder is placed in the high word. The sign of the remainder is always the same as the sign of the dividend, unless the remainder is zero. If the quotient will not fit in 16 bits, the V (overflow) condition code bit is set and the data register remains unmodified.

If the effective address operand is zero, the processor takes a Division by Zero exception through vector 5.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	0			•			•	•	•	•

X Not affected.
N Set if quotient is negative (bit 15 = 1). Cleared otherwise. Undefined on divide by zero or overflow.
Z Set if quotient is zero. Cleared otherwise. Undefined on divide by zero or overflow.
V Set if an overflow occurs. Cleared otherwise. Undefined on divide by zero.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An	•	
(An)+	011	An	•	
-(An)	100	An	•	
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	110	An	•	
([bd,An],Xn,od)	110	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100	•	
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Register			1	1	1	Mode			Register		

Example The DIVS.W D1,D0 instruction divides the longword quantity in D0 by the low-order 16 bits in D1. The quotient and remainder are placed in the low and high words of D0, respectively. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Register			1	1	1	Mode			Register		
1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1

The instruction has hex value $81C1. Execution yields:

Before:

```
USP=b03f8aa0 MSP=b03f8aa0 ISP =b03f8aa0 SR =2700 -- -- SI PL=7 -----
D0: fffffffd fffffffe 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8aa0
PC =DIVS$    DIVS.W   D1,D0
```

After:

```
USP=b03f8aa0 MSP=b03f8aa0 ISP =b03f8aa0 SR =2700 -- -- SI PL=7 -----
D0: ffff0001 fffffffe 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8aa0
PC =DIVS$+002         NOP
```

This example divides -3 (in D0) by -2 (in D1), leaving a quotient of 1 and remainder of -1.

DIVSL/DIVS.L Instructions

Signed Divide (Long Forms)

Syntax: DIVS.L <ea>,Dq
DIVS.L <ea>,Dr:Dq
DIVSL.L <ea>,Dr:Dq

Description

The 68020 and 68030 support divide instructions that allow 32-bit division. There are three forms of 32-bit signed division:

DIVS.L	**<ea>,Dq**	32-bit dividend / 32-bit divisor yielding 32-bit quotient.
DIVS.L	**<ea>,Dr:Dq**	64-bit dividend / 32-bit divisor yielding 32-bit quotient and 32-bit remainder.
DIVSL.L	**<ea>,Dr:Dq**	32-bit dividend / 32-bit divisor yielding 32-bit quotient and 32-bit remainder.

The effective address operand is always the divisor. The quotient is in the Dq data register for the first two forms and in the Dr:Dq pair for the 64-bit form. Registers Dr and Dq receive the remainder and quotient, respectively. The dividend operand is lost. The sign of the remainder is always the same as the sign of the dividend, unless the remainder is zero. If the quotient will not fit in 32 bits, the V (overflow) condition code bit is set and the data register remains unmodified.

If the effective address operand is zero, the processor takes a Division by Zero exception through vector 5.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	0				•				•	•

X	Not affected.
N	Set if quotient is negative (bit 31 = 1). Cleared otherwise. Undefined on divide by zero or overflow.
Z	Set if quotient is zero. Cleared otherwise. Undefined on divide by zero or overflow.
V	Set if an overflow occurs. Cleared otherwise. Undefined on divide by zero.
C	Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An	•	
(An)+	011	An	•	
-(An)	100	An	•	
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	110	An	•	
([bd,An],Xn,od)	110	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100	•	
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	1	Mode			Register		
0	Register Dq			1	Size	0	0	0	0	0	0	0	Register Dr		

0 => 32-bit form. Dividend is in Dq
1 => 64-bit form. Dividend is in Dq:Dr

The remainder is discarded by specifying the same register for Dq and Dr. This is possible only for the 32-bit form.

Example The DIVS.L D2,D0:D1 instruction divides the quadword (64-bit) quantity in D0:D1 by the longword quantity in D2. The quotient and remainder are placed in the D1 and D0, respectively. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	1	Mode			Register		
0	Register Dq			1	Size	0	0	0	0	0	0	0	Register Dr		

0	1	0	0	1	1	0	0	0	1	0	0	0	0	1	0
0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0

The instruction has hex value $4C42 1C00. Execution yields:

Before:

```
USP=b03f8a84 MSP=b03f8a84 ISP =b03f8a84 SR =2700 -- -- SI PL=7 -----
D0: ffffffff fffffffd fffffffe 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a84
PC =DIVSL$    DIVS.L   D2,D0:D1
```

After:

```
USP=b03f8a84 MSP=b03f8a84 ISP =b03f8a84 SR =2700 -- -- SI PL=7 -----
D0: ffffffff 00000001 fffffffe 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a84
PC =DIVSL$+004         NOP
```

This example divides -3 (in D0:D1) by -2 (in D2), leaving a quotient of 1 (in D1, and a remainder of -1 (in D0).

DIVU Instruction

Unsigned Divide (Word Form)

Syntax: DIVU.W <ea>,Dn

Description

The DIVU.W instruction divides a 32-bit quantity by a 16-bit quantity, yielding a 16-bit quotient and a 16-bit remainder. The division is unsigned (both operands are positive). The effective address operand is always the divisor, and the data register is always the dividend. The quotient is placed in the low word of the data register, and the remainder is placed in the high word. If the quotient will not fit in 16 bits, the V (overflow) condition code bit is set, and the data register remains unmodified.

If the effective address operand is zero, the processor takes a Division by Zero exception through vector 5.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	0			•			•	•	•	•

X Not affected.
N Set if quotient is negative (bit 15 = 1). Cleared otherwise. Undefined on divide by zero or overflow.
Z Set if quotient is zero. Cleared otherwise. Undefined on divide by zero or overflow.
V Set if an overflow occurs. Cleared otherwise. Undefined on divide by zero.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000	•	
An	001	An			xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100	•	
(An)+	011	An	•						
-(An)	100	An	•						
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Register			0	1	1	Mode			Register		

Example The DIVU.W D1,D0 instruction divides the longword quantity in D0 by the low-order 16 bits in D1. The quotient and remainder are placed in the low and high words of D0, respectively. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Register			0	1	1	Mode			Register		
1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1

The instruction has hex value $80C1. Execution yields:

Before:

```
USP=b03f8a74 MSP=b03f8a74 ISP =b03f8a74 SR =2700 -- -- SI PL=7 -----
D0: 00080001 00008000 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a74
PC =DIVU$    DIVU.W   D1,D0
```

After:

```
USP=b03f8a74 MSP=b03f8a74 ISP =b03f8a74 SR =2700 -- -- SI PL=7 -----
D0: 00010010 00008000 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a74
PC =DIVU$+002         NOP
```

This example divides hex $80001 (in D0) by hex $8000 (in D1), leaving a quotient of hex $10 and a remainder of 1.

DIVUL/DIVU.L Instructions

Unsigned Divide (Long Forms)

Syntax:
DIVU.L <ea>,Dq
DIVU.L <ea>,Dr:Dq
DIVUL.L <ea>,Dr:Dq

Description

The 68020 and 68030 support divide instructions that allow 32-bit division. There are three forms of 32-bit signed division:

DIVU.L	**<ea>,Dq**	32-bit dividend / 32-bit divisor yielding 32-bit quotient.
DIVU.L	**<ea>,Dr:Dq**	64-bit dividend / 32-bit divisor yielding 32-bit quotient and 32-bit remainder.
DIVUL.L	**<ea>,Dr:Dq**	32-bit dividend / 32-bit divisor yielding 32-bit quotient and 32-bit remainder.

All operands are treated as unsigned (positive) quantities. The effective address operand is always the divisor. The quotient is in the Dq data register for the first two forms and in the Dr:Dq pair for the 64-bit form. Registers Dr and Dq receive the remainder and quotient, respectively. The dividend operand is lost. The sign of the remainder is always the same as the sign of the dividend, unless the remainder is zero. If the quotient will not fit in 32 bits, the V (overflow) condition code bit is set and the data register remains unmodified.

If the effective address operand is zero, the processor takes a Division by Zero exception through vector 5.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	0				•				•	•

X Not affected.
N Set if quotient is negative (bit 63/31 = 1). Cleared otherwise. Undefined on divide by zero or overflow.
Z Set if quotient is zero. Cleared otherwise. Undefined on divide by zero or overflow.
V Set if an overflow occurs. Cleared otherwise. Undefined on divide by zero.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An	•	
(An)+	011	An	•	
-(An)	100	An	•	
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	110	An	•	
([bd,An],Xn,od)	110	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100	•	
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="3">Mode</td><td colspan="3">Register</td></tr>
<tr><td>0</td><td colspan="3">Register Dq</td><td>0</td><td>Size</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="3">Register Dr</td></tr>
</table>

0 => 32-bit form. Dividend is in Dq
1 => 64-bit form. Dividend is in Dq:Dr

The remainder is discarded by specifying the same register for Dq and Dr. This is possible only for the 32-bit form.

Example The DIVU.L D2,D0:D1 instruction divides the quadword (64-bit) quantity in D0:D1 by the longword quantity in D2. The quotient and remainder are placed in D1 and D0, respectively. This instruction has the following layout:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="3">Mode</td><td colspan="3">Register</td></tr>
<tr><td>0</td><td colspan="3">Register Dq</td><td>0</td><td>Size</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="3">Register Dr</td></tr>
</table>

0	1	0	0	1	1	0	0	0	1	0	0	0	0	1	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0

The instruction has hex value $4C42 1400. Execution yields:

Before:

```
USP=b03f8a70 MSP=b03f8a70 ISP =b03f8a70 SR =2700 -- -- SI PL=7 -----
D0: 70000001 12345678 80000000 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a70
PC =DIVUL$    DIVU.L   D2,D0:D1
```

After:

```
USP=b03f8a70 MSP=b03f8a70 ISP =b03f8a70 SR =2708 -- -- SI PL=7 -N---
D0: 12345678 e0000002 80000000 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a70
PC =DIVUL$+004         NOP
```

This example divides the hex quantity $70000001 12345678 (in D0:D1) by hex $80000000 (in D2), leaving a quotient of hex $e0000002 (in D1) and a remainder of hex $12345678 (in D0).

EOR Instruction

Syntax: EOR Dn,<ea>

Exclusive OR Logical

Description

The EOR instruction performs an exclusive OR operation between a data register and an effective address operand. The data register must be the source and the effective address must be the destination.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0		•	•	•		•	•	•	•

X Not affected.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100		
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	(bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

If you specify an immediate mode source (instead of a data register), many assemblers will generate an EORI instruction.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	Data Register			1	Size		Mode			Register		

00=Byte
01=Word
10=Long

Example The EOR.L D0,D1 instruction performs a 32-bit logical exclusive OR between the contents of D0 and the contents of D1.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	Data Register			1	Size		Mode			Register		
1	0	1	1	0	0	0	1	1	0	0	0	0	0	0	1

The result is left in D1. This instruction has the following layout:

The hex value of the instruction is $B181. Instruction execution yields:

Before:

```
USP=b03f8a36 MSP=b03f8a36 ISP =b03f8a36 SR =2700 -- -- SI PL=7 -----
D0: 98765432 fec80123 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a36
PC =EOR$        EOR.L      D0,D1
```

After:

```
USP=b03f8a36 MSP=b03f8a36 ISP =b03f8a36 SR =2700 -- -- SI PL=7 -----
D0: 98765432 66be5511 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a36
PC =EOR$+002    NOP
```

EORI Instruction

Exclusive OR Immediate

Syntax: EORI #<data>,<ea>

Description

The EORI instruction performs an exclusive OR operation on an effective address operand using immediate data.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0		•	•	•		•	•	•	•

X Not affected.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100	•	
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td colspan="2">Size</td><td colspan="3">Mode</td><td colspan="3">Register</td></tr>
<tr><td colspan="8">Word Data (16 bits)</td><td colspan="8">Byte Data (8 bits)</td></tr>
<tr><td colspan="16">Long Data</td></tr>
</table>

The size field indicates the operation size: 00 for byte, 01 for word, and 10 for long. There are 2 or 4 bytes of immediate data that follow the instruction in memory. Byte and word operations have 16 bits (2 bytes) of immediate data. Long operations have 32 bits (4 bytes).

Example The instruction EORI.L #$FEC80123,D0 performs an Exclusive OR between the 32-bit value in register D0 with the hex value $FEC80123. The immediate data is contained in the two words following the instruction, as shown below:

Instruction

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td colspan="2">Size</td><td colspan="3">Mode</td><td colspan="3">Register</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Data

1	1	1	1	1	1	1	0	1	1	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	1

This instruction has hex value $0A80 FEC8 0123. Executing the instruction yields:

Before:

```
USP=b03f8a36 MSP=b03f8a36 ISP =b03f8a36 SR =2700 -- -- SI PL=7 -----
D0: 98765432 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a36
PC =EORI$     EORI.L   #$FEC80123,D0
```

After:

```
USP=b03f8a36 MSP=b03f8a36 ISP =b03f8a36 SR =2700 -- -- SI PL=7 -----
D0: 66be5511 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a36
PC =EORI$+006          NOP
```

EORI to CCR Instruction Syntax: EORI #<data>,CCR

Exclusive OR Immediate to Condition Code Register

Description

The EORI to CCR instruction performs a logical Exclusive OR operation on the Condition Code Register using an 8-bit immediate field. This is useful for inverting any arbitrary combination of condition code bits.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•				•	•	•	•

X Inverted if bit 4 of immediate operand is one. Unchanged otherwise.
N Inverted if bit 3 of immediate operand is one. Unchanged otherwise.
Z Inverted if bit 2 of immediate operand is one. Unchanged otherwise.
V Inverted if bit 1 of immediate operand is one. Unchanged otherwise.
C Inverted if bit 0 of immediate operand is one. Unchanged otherwise.

Instruction Format

0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	Byte Data (8 bits)							

Example The EORI #$0E,CCR instruction inverts the N, Z, and V condition code bits. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

The instruction has hex value $0A3C 000E. Execution yields:

Before:

```
USP=b03f8a36 MSP=b03f8a36 ISP =b03f8a36 SR =271f -- -- SI PL=7 XNZVC
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a36
PC =EORICCR$    EORI.B   #$0E,CCR
```

After:

```
USP=b03f8a36 MSP=b03f8a36 ISP =b03f8a36 SR =2711 -- -- SI PL=7 X---C
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a36
PC =EORICCR$+004         NOP
```

EORI to SR Instruction (Privileged) Syntax: EORI #<data>,SR

Exclusive OR Immediate to Status Register

Description

The EORI to SR instruction performs an Exclusive OR operation on the Status Register using a 16-bit immediate field. This is useful for inverting any arbitrary combination of bits in this register. This processor must be in Supervisor mode to execute this instruction. Attempted execution in user mode will cause a Privilege Violation exception through vector 8.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*			•			•	•	•	•

X Inverted if bit 4 of immediate operand is one. Unchanged otherwise.
N Inverted if bit 3 of immediate operand is one. Unchanged otherwise.
Z Inverted if bit 2 of immediate operand is one. Unchanged otherwise.
V Inverted if bit 1 of immediate operand is one. Unchanged otherwise.
C Inverted if bit 0 of immediate operand is one. Unchanged otherwise.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0
Word Data (16 bits)															

Example The EORI #$2413,SR instruction inverts the X, V, and C condition code bits, as well as the Supervisor status bit, and the high bit of the interrupt mask. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0
0	0	1	0	0	1	0	0	0	0	0	1	0	0	1	1

The instruction has hex value $0A7C 2413. Execution yields:

Before:

```
USP=b03f8a36 MSP=b03f8a36 ISP =b03f8a36 SR =271f -- -- SI PL=7 XNZVC
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a36
PC =EORISR$     EORI.W   #$2413,SR
```

After:

```
USP=b03f8a36 MSP=b03f8a36 ISP =b03f8a36 SR =030c -- -- UI PL=3 -NZ--
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8a36
PC =EORISR$+004          NOP
```

EXG Instruction

Exchange Registers

Syntax: EXG Dx,Dy
EXG Ax,Ay
EXG Dx,Ay

Description

The EXG instruction allows you to exchange the entire contents of two registers. You can exchange two address registers, two data registers, or an address register with a data register.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-				•		•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

Exchanging Two Data Registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Register Dx			1	0	1	0	0	0	Register Dy		

Exchanging Two Address Registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Register Ax			1	0	1	0	0	1	Register Ay		

Exchanging a Data Register and an Address Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Register Dx			1	1	0	0	0	1	Register Ay		

When exchanging an address register and a data register, the address register must be specified in bits 2–0.

Example The EXG D0,A0 instruction exchanges the contents of register D0 and Register A0. This instruction uses the third format as shown above, with the following contents:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Register Dx			1	1	0	0	0	1	Register Ay		
1	1	0	0	0	0	0	1	1	0	0	0	1	0	0	0

The hex value of the instruction is $C188. Execution yields:

Before:

```
USP=b03f89f8 MSP=b03f89f8 ISP =b03f89f8 SR =2700 -- -- SI PL=7 -----
D0: 11111111 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 22222222 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f89f8
PC =EXG$        EXG         D0,A0
```

After:

```
USP=b03f89f8 MSP=b03f89f8 ISP =b03f89f8 SR =2700 -- -- SI PL=7 -----
D0: 22222222 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 11111111 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f89f8
PC =EXG$+002    NOP
```

EXT Instruction

Sign Extend (Byte->Word and Word->Long)

Syntax: **EXT.W** **Dx**
EXT.L **DX**

Description

The EXT instruction performs a sign extension operation on a data register. You can extend an 8-bit quantity to 16 bits, or a 16-bit quantity to 32 bits. Extending a byte to a word replicates bit 7 in bits 8–15 of the data register. Extending a word quantity to a long replicates bit 15 in bits 16–31 of the data register.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0			•	•		•	•	•	•

X Not affected.
N Set if the result is negative (bit 31 or bit 15 is a one). Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	Size		0	0	0	Data Register		

10 => Extend byte to word
11 => Extend word to long

Example The EXT.L D0 instruction replicates bit 15 of register D0 in bits 16–31 of this register. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	Size		0	0	0	Data Register		
0	1	0	0	1	0	0	0	1	1	0	0	0	0	0	0

The hex value of the instruction is $48C0. Execution yields:

Before:

```
USP=b03f89f8 MSP=b03f89f8 ISP =b03f89f8 SR =2700 -- -- SI PL=7 -----
D0: 12348765 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f89f8
PC =EXT$        EXT.L      D0
```

After:

```
USP=b03f89f8 MSP=b03f89f8 ISP =b03f89f8 SR =2708 -- -- SI PL=7 -N---
D0: ffff8765 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f89f8
PC =EXT$+002    NOP
```

EXTB Instruction

Syntax: EXTB.L Dx

Sign Extend (Byte->Long)

Description

The EXTB instruction performs a sign extension operation on a data register, replicating bit 7 of the register in bits 8–31.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0				•				•	•

X Not affected.
N Set if the result is negative (bit 31 is a one). Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	1	1	1	0	0	0	Data Register		

Example The EXTB.L D0 instruction replicates bit 7 of register D0 in bits 8–31 of this register. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	1	1	1	0	0	0	Data Register		
0	1	0	0	1	0	0	1	1	1	0	0	0	0	0	0

The hex value of the instruction is $49C0. Execution yields:

Before:

```
USP=b03f89f8 MSP=b03f89f8 ISP =b03f89f8 SR =2700 -- -- SI PL=7 -----
D0: 00000081 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f89f8
PC =EXTB$     EXTB.L   D0
```

After:

```
USP=b03f89f8 MSP=b03f89f8 ISP =b03f89f8 SR =2708 -- -- SI PL=7 -N---
D0: ffffff81 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f89f8
PC =EXTB$+002          NOP
```

Illegal Instruction

Syntax: ILLEGAL

Take Illegal Instruction Exception

Description

The Illegal instruction causes an illegal instruction exception through vector 4. The exception generates a format 0 stack frame shown in Figure 2–34. This instruction is guaranteed to produce an illegal instruction exception on all 68000 family members.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-						•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	1	1	1	1	0	0

Example The hex value of the instruction is $4AFC. Execution yields:

Before:

```
USP=b03f89f8 MSP=b03f89f8 ISP =b03f89f8 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f89f8
PC =ILLEGAL$    ILLEGAL
```

After:

```
Illegal Instruction
USP=b03f89f8 MSP=b03f89f8 ISP =b03f89f8 SR =a700 T1 -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f89f8
PC =ILLEGAL$    ILLEGAL
```

Note that the PC does not advance as a result of instruction execution. The instruction trace bit in the stack frame is also not cleared by the exception.

JMP Instruction

Jump to Effective Address

Syntax: JMP <ea>

Description

The JMP instruction evaluates an effective address and loads this address into the PC. This instruction is similar to a BRA instruction, except that the address need not be PC-relative.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-						•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn			xxx.W	111	000	•	
An	001	An			xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100		
(An)+	011	An							
-(An)	100	An							
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	1	Mode			Data Register		

Example The JMP (A0) instruction branches to the address contained in register A0. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	1	Mode			Register		
0	1	0	0	1	1	1	0	1	1	0	1	0	0	0	0

The hex value of this instruction is $4ED0. Execution yields:

Before:

```
USP=b03f89ac MSP=b03f89ac ISP =b03f89ac SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8f7a 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f89ac
PC =JMP$       JMP       (A0)
```

After:

```
USP=b03f89ac MSP=b03f89ac ISP =b03f89ac SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8f7a 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f89ac
PC =b03f8f7a   NOP
```

JSR Instruction

Jump to Subroutine at Effective Address

Syntax: JSR <ea>

Description

The JSR instruction evaluates an effective address and loads this address into the PC. The address of the instruction following the JSR is pushed onto the stack. This instruction is similar to a BSR instruction, except that the address need not be PC-relative.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-						•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		
An	001	An		
(An)	010	An	•	
(An)+	011	An		
-(An)	100	An		
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	110	An	•	
([bd,An],Xn,od)	110	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100		
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	0	Mode			Data Register		

Example The JSR (A0) instruction calls the subroutine whose address is contained in register A0. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	0	Mode			Register		
0	1	0	0	1	1	1	0	1	0	0	1	0	0	0	0

The hex value of this instruction is $4E90. Execution yields:

Before:

```
USP=b03f89ac MSP=b03f89ac ISP =b03f89ac SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8fa6 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f89ac
PC =JSR$        JSR        (A0)
```

After:

```
USP=b03f89ac MSP=b03f89ac ISP =b03f89a8 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8fa6 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f89a8
PC =b03f8fa6    NOP
```

LEA Instruction

Load Effective Address

Syntax: LEA <ea>,An

Description

The LEA instruction evaluates an effective address and loads this address into the specified address register.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-				•		•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		
An	001	An		•
(An)	010	An	•	
(An)+	011	An		
-(An)	100	An		
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	110	An	•	
([bd,An],Xn,od)	110	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100		
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	Register			1	1	1	Mode			Register		

Example The LEA 4(A0),A0 instruction adds 4 to the contents of register A0. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	Register			1	1	1	Mode			Register		
0	1	0	0	0	0	0	1	1	1	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

The hex value of this instruction is $41E8 0004. Execution yields:

Before:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 00000042 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =LEA$        LEA         $0004(A0),A0
```

After:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 00000046 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =LEA$+004    NOP
```

LINK Instruction

Link and Allocate

Syntax: LINK An,#size

Description

The LINK instruction pushes the specified address register on the stack, then adds the immediate operand to the stack pointer. The address register is loaded with the address of the old contents of the register. On the 68000 and 68010, the size is limited to 16 bits. On the 68020 and 68030, the immediate value may be either 16 or 32 bits. The LINK instruction is equivalent to the following sequence:

```
MOVE.L      An,-(SP)    ; Save old An contents
MOVEA.L     SP,An       ; An -> old contents of An
ADD.[W/L]   #size,SP    ; Allocate temporary space
```

This instruction is normally used to allocate stack space for local variables in a procedure. The immediate operand is normally negative, and the stack grows by the size specified plus 4 bytes for the saved contents of the address register. Locations in the allocated space (called a *stack frame*) are at a constant (negative) offset from the address register. The LINK instruction is normally the first instruction in a procedure. The procedure arguments are at positive offsets from the address register.

Condition Codes	X	N	Z	V	C	**Sizes**	Byte	Word	Long	**CPUs**	000	010	020	030
	-	-	-	-	-			•	•		•	•	•	•

☐ 68020/030 only

X	Not affected.
N	Not affected.
Z	Not affected.
V	Not affected.
C	Not affected.

Instruction Format

Word Form

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	Register An		
16-bit Stack Frame Size															

Long Form

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	0	1	Register An		
32-bit Stack Frame Size (High Word)															
32-bit Stack Frame Size (Low Word)															

Example The LINK A6,#-32 instruction creates a 32-byte stack frame using register A6. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	Register An		
16-bit Stack Frame Size															

0	1	0	0	1	1	1	0	0	1	0	1	0	1	1	0
0	1	1	1	1	1	1	1	1	1	0	0	1	1	1	0

The hex value of this instruction is $4E56 FFCE. Execution yields:

Before:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =LINK$       LINK       A6,#$FFCE
```

After:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f88de SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 b03f8910 b03f88de
PC =LINK$+004   NOP
```

Note that register A7 was decremented by 36 (hex $24) bytes (32 for the stack frame and 4 for the old contents of A6).

LSL Instruction
Logical Shift Left

Syntax:

LSL	**Dx,Dy**
LSL	**#<data>,Dy**
LSL	**<ea>**

Description

The LSL instruction performs a logical shift left on a data register or on an effective address operand. The shift count for a data register can be specified one of two ways: The low 5 bits of a data register can be used for the count value (LSL Dx,Dy form), or a small constant in the instruction itself can hold the count (LSL #<data>,Dy form). The register form allows shifts of up to 63 bits, while the immediate constant can have a value in the range 1–8. The data register form of the instruction supports all sizes: byte, word, and longword.

The effective address form shifts only 1 bit and supports only the word data size.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	0	*		●	●	●		●	●	●	●

☐ Register form only

X Set to the last bit shifted out of the operand. Unaffected for a shift count of zero.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Set to the last bit shifted out of the operand. Cleared for a shift count of zero.

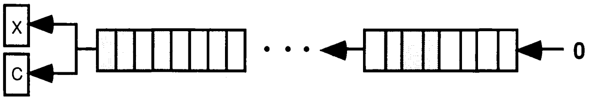
Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100	•	
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

☐ 68020/030 only ☐ Register form only

Bits shifted out of the most significant bit (31/15/7) are placed in both the Extend (X) and Carry (C) condition code bits, as shown below. Zeros are shifted in from the right. This operation is effectively a multiplication by a power of two (the power is the number of bits shifted).



Instruction Format

Register Shifts

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			1	Size		i/r	0	1	Register		

Size:
00=Byte
01=Word
10=Long

i/r:
0 = Immediate Shift count
1 = Register Shift count

The Count/Register field contains either a register number (bit 5 = 1) or an immediate shift count (bit 5 = 0). The immediate shift count form uses bits 11–9 to hold a value from 1–7. A value of zero in these bits represents a value of 8. Bits 2–0 contain the number of the register to be shifted.

Memory Shifts

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	1	1	1	Mode			Register		

Bit 8 in both formats indicates the shift direction. A one indicates a left shift, and zero indicates a right shift.

Example The LSL.L D1,D0 instruction shifts the 32 bit quantity in D0 left by the number of bits specified by register D1. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			1	Size		i/r	0	1	Register		
1	1	1	0	0	0	1	1	1	0	1	0	1	0	0	0

The instruction has hex value $E3A8. Execution yields:

Before:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 87654321 00000013 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =LSL$     LSL.L    D1,D0
```

After:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 19080000 00000013 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =LSL$+002          NOP
```

In this example, the shift count is $13 (19 decimal).

LSR Instruction

Logical Shift Right

Syntax: **LSR Dx,Dy**
LSR #<data>,Dy
LSR <ea>

Description

The LSR instruction performs a logical shift right on a data register or an effective address operand. The shift count for a data register can be specified one of two ways: The low 5 bits of a data register can be used for the count value (LSR Dx,Dy form), or a small constant in the instruction itself can hold the count (LSR #<data>,Dy form). The register form allows shifts of up to 63 bits, while the immediate constant can have a value in the range 1–8. The data register form of the instruction supports all sizes: byte, word, and longword.

The effective address form shifts only 1 bit and supports only the word data size.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•	•	•		•	•	•	•

☐ Register form only

X Set to the last bit shifted out of the operand. Unaffected for a shift count of zero.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Set if the most significant bit changes at any time during the operation. Cleared otherwise.
C Set to the last bit shifted out of the operand. Cleared for a shift count of zero.

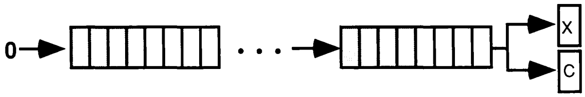
The N bit will only become set if the original destination is negative and the shift count is zero.

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100	•	
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

☐ 68020/030 only ☐ Register form only

Bits shifted out of bit 0 are placed in both the Extend (X) and Carry (C) condition code bits, as shown below. Zeros are shifted into the most significant bit (31/15/7) so that the number is always considered as a positive quantity. This operation is effectively an unsigned division by a power of two (the power is the number of bits shifted).



Instruction Format

Register Shifts

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			0	Size		i/r	0	1	Register		

00=Byte
01=Word
10=Long

0 = Immediate Shift count
1 = Register Shift count

The Count/Register field contains either a register number (bit 5 = 1) or an immediate shift count (bit 5 = 0). The immediate shift count form uses bits 11–9 to hold a value from 1–7. A value of zero in these bits represents a value of 8. Bits 2–0 contain the number of the register to be shifted.

Memory Shifts

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	0	1	1	Mode			Data Register		

Bit 8 in both formats indicates the shift direction. A one indicates a left shift, and zero indicates a right shift.

Example The LSR.L D1,D0 instruction shifts the 32 bit quantity in D0 right by the number of bits specified by register D1. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			0	Size		i/r	0	1	Register		
1	1	1	0	0	0	1	0	1	0	1	0	1	0	0	0

The instruction has hex value $E2A8. Execution yields:

Before:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 87654321 00000013 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =LSR$       LSR.L      D1,D0
```

After:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2711 -- -- SI PL=7 X---C
D0: 000010ec 00000013 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =LSR$+002    NOP
```

In this example, the shift count is $13 (19 decimal). The last bit shifted out of bit 0 of D0 was a one, setting both the X and C condition code bits.

MOVE Instruction

Syntax: MOVE <ea>,<ea>

MOVE Data from Source to Destination

Description

The MOVE instruction copies data from one effective address operand to another. The condition codes are set according to this data.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0		•	•	•		•	•	•	•

X Not affected.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

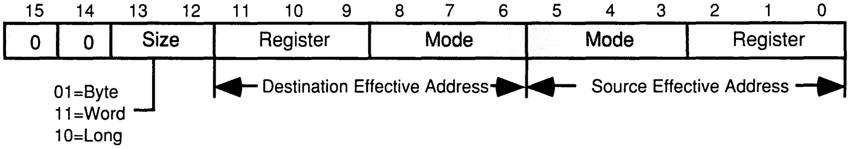
Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000	•	•
An	001	An	•		xxx.L	111	001	•	•
(An)	010	An	•	•	#<data>	111	100	•	
(An)+	011	An	•	•					
-(An)	100	An	•	•					
(d16,An)	101	An	•	•	(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•	•	(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•	•	(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•	•	([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•	•	([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only ☐ Word and long sizes only

Most assemblers will generate a MOVEA instruction when MOVE is specified with an address register destination. Many assemblers will also change an immediate mode MOVE to a data register to a MOVEQ if the immediate data is an 8-bit constant.

Instruction Format


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0
0
Size
Register
Mode
Mode
Register
01=Byte
11=Word
10=Long
Destination Effective Address
Source Effective Address

Example The MOVE.L D0,D1 instruction copies the contents of register D0 into register D1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	Size		Register			Mode			Mode			Register		
0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0

The instruction has hex value $2200. Execution yields:

Before:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 87654321 12345678 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVE$    MOVE.L   D0,D1
```

After:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2708 -- -- SI PL=7 -N---
D0: 87654321 87654321 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVE$+002         NOP
```

The N condition code bit was set because the source data was negative.

MOVEA Instruction

MOVE Data from Source to Address Register

Syntax: MOVEA <ea>,An

Description

The MOVEA instruction copies data from an effective address operand to an address register. The condition codes are unaffected by this instruction.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-			•	•		•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

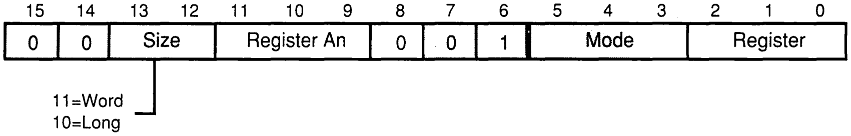
Word data is sign-extended to 32-bits before use. The entire address register is always modified, regardless of operation size.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•		xxx.W	111	000	•	
An	001	An	•	•	xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100	•	
(An)+	011	An	•						
-(An)	100	An	•						
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

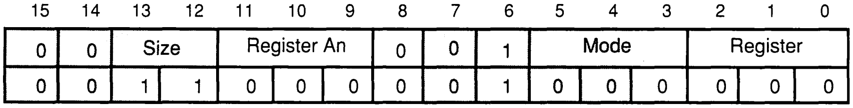
☐ 68020/030 only

Instruction Format



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	Size		Register An			0	0	1	Mode			Register		

Example The MOVEA.W D0,A0 instruction sign-extends the low-order 16 bits of register D0 into register A0. This instruction has the following layout:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	Size		Register An			0	0	1	Mode			Register		
0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0

The instruction has hex value $3040. Execution yields:

Before:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 00008000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVEA$      MOVEA.W    D0,A0
```

After:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 00008000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: ffff8000 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVEA$+002  NOP
```

MOVE From CCR Instruction Syntax: MOVE CCR,<ea>

MOVE Data from Condition Code Register to Effective Address

Description

The MOVE from CCR instruction copies the low-order 8 bits of the Status Register into an effective address. The condition codes are unaffected by this instruction. MOVE from CCR fixes a design flaw in the original 68000 chip. On the 68000, user mode programs could read the status register, including the high byte. There was no instruction to read only the condition codes. Many programs (including some compilers) used the MOVE from SR instruction for this purpose. Adding MOVE from CCR and making MOVE to SR a privileged instruction prevents user mode programs from reading the system byte of the status register, but the 68000 is now incompatible with the 68010 and later chips.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-			•				•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

MOVE from CCR is always a word operation, with the upper 8 bits zero.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100		
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	1	Mode			Register		

Example The MOVE CCR,D0 instruction copies the CCR into the low-order 16 bits of register D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	1	Mode			Register		
0	1	0	0	0	0	1	0	1	1	0	0	0	0	0	0

The instruction has hex value $42C0. Execution yields:

Before:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =271f -- -- SI PL=7 XNZVC
D0: 0000ff00 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVEFCCR$    MOVE    CCR,D0
```

After:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =271f -- -- SI PL=7 XNZVC
D0: 0000001f 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVEFCCR$+002        NOP
```

MOVE To CCR Instruction

Syntax: MOVE <ea>,CCR

MOVE Data to Condition Code Register from Effective Address

Description

The MOVE to CCR instruction copies the low-order 8 bits of the effective address operand into the low 8 bits of the Status Register. MOVE to CCR is always a word operation. However, the upper 8 bits of the source operand are ignored, and the upper 8 bits of the Status Register are unaffected. The condition codes are set to the contents of the source operand.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*			•				•	•	•

X Set according to bit 4 of the source operand.
N Set according to bit 3 of the source operand.
Z Set according to bit 2 of the source operand.
V Set according to bit 1 of the source operand.
C Set according to bit 0 of the source operand.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•		xxx.W	111	000	•	
An	001	An			xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100	•	
(An)+	011	An	•						
-(An)	100	An	•						
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	Mode			Register		

Example The MOVE D0,CCR instruction copies the low-order 8 bits of register D0 into the CCR. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	Mode			Register		
0	1	0	0	0	1	0	0	1	1	0	0	0	0	0	0

The instruction has hex value $44C0. Execution yields:

Before:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 0000001f 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVETCCR$   MOVE       D0,CCR
```

After:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =271f -- -- SI PL=7 XNZVC
D0: 0000001f 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVETCCR$+002          NOP
```

MOVE From SR Instruction (Privileged) Syntax: MOVE SR,<ea>

MOVE Data from Status Register to Effective Address

Description

The MOVE from SR instruction copies the Status Register into an effective address. The condition codes are unaffected by this instruction. This instruction is privileged, except on the 68000 chip. Execution of a MOVE from SR instruction in user mode will cause a Privilege Violation exception through vector 8 on the 68010, 68020, and 68030.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-			•			•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•
An	001	An		
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100		
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Mode			Register		

Example The MOVE SR,D0 instruction copies the Status Register into the low-order 16 bits of register D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Mode			Register		
0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0

The instruction has hex value $40C0. Execution yields:

Before:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =271f -- -- SI PL=7 XNZVC
D0: 0000ff00 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVEFSR$    MOVE     SR,D0
```

After:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =271f -- -- SI PL=7 XNZVC
D0: 0000a71f 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVEFSR$+002         NOP
```

Note that the trace bit (T1) is set in the destination operand. This is a result of running the instruction in trace mode to obtain the execution data.

Note

The 68000 chip reads a memory destination before writing it. There are two bus cycles (a read and a write) to the destination instead of the single bus cycle you would expect. The 68010, 68020, and 68030 do not exhibit this behavior.

MOVE To SR Instruction (Privileged) Syntax: MOVE <ea>,SR

MOVE Data to Status Register from Effective Address

Description

The MOVE to SR instruction copies the 16-bit effective address operand into the Status Register. The condition codes are set to the contents of the source operand.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*			•			•	•	•	•

X Set according to bit 4 of the source operand.
N Set according to bit 3 of the source operand.
Z Set according to bit 2 of the source operand.
V Set according to bit 1 of the source operand.
C Set according to bit 0 of the source operand.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•		xxx.W	111	000	•	
An	001	An			xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100	•	
(An)+	011	An	•						
-(An)	100	An	•						
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	1	1	Mode			Register		

Example The MOVE D0,SR instruction copies the low-order 16 bits of register D0 into the Status Register. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	1	1	Mode			Register		
0	1	0	0	0	1	1	0	1	1	0	0	0	0	0	0

The instruction has hex value $46C0. Execution yields:

Before:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 0000271f 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVETSR$    MOVE     D0,SR
```

After:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =271f -- -- SI PL=7 XNZVC
D0: 0000271f 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVETSR$+002         NOP
```

MOVE USP Instruction (Privileged)

Syntax: MOVE USP,An
MOVE An,USP

Move User Stack Pointer

Description

The MOVE USP instruction allows a supervisor mode program to fetch and set the user mode stack pointer. This instruction is required to allow the supervisor mode program to initialize, save, and restore the user program stack area. MOVE USP is vital to multi-tasking support and for loading user mode programs.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-				•		•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	0	Dir	Register An		

0=> An to USP
1=> USP to An

Example The MOVE USP,A0 instruction copies the user mode stack pointer into register A0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	0	Dir	Register An		
0	1	0	0	1	1	1	0	0	1	1	0	1	0	0	0

The hex value of this instruction is $4E68. Execution yields:

Before:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVEUSP$    MOVE     USP,A0
```

After:

```
USP=b03f8914 MSP=b03f8914 ISP =b03f8914 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8914 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8914
PC =MOVEUSP$+002         NOP
```

MOVEC Instruction (Privileged)

Move Control Register

Syntax: MOVEC Rc,Rn
MOVEC Rn,Rc

Description

The MOVEC instruction allows a supervisor mode program to fetch and set 68010, 68020, and 68030 control registers. This instruction permits changing the chips' internal registers for configuration purposes.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-				•			•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	Dir
D/A	Register Rn			Control Register											

Bit 0 of the first instruction word gives the transfer direction. This bit is a zero when moving from the control register to the general register and one to move the general register to the control register. Bits 15–12 of the second instruction word identify the general register. Bits 14–12 contain the register number and Bit 15 is a one to indicate an address register and a zero to indicate a data register.

Bits 11–0 of the second instruction word specify the control register. The values allowed for this field are

Hex	Control Register
000	Source Function Code Register (SFC)
001	Destination Function Code Register (DFC)
002	CAche Control Register (CACR) (Not on 68010)
800	User Stack Pointer (USP)
801	Vector Base Register (VBR)
802	CAche Address Register (CAAR) (Not on 68010)
803	Master Stack Pointer (MSP) (Not on 68010)
804	Interrupt Stack Pointer (Not on 68010)

Example The MOVEC MSP,D0 instruction copies the master stack pointer into register D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	Dir
D/A	Register Rn			Control Register											

0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1

The hex value of this instruction is $4E7A 0803. Execution yields:

Before:

```
USP=b03f8910 MSP=b03f8914 ISP =b03f8918 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8918
PC =MOVEC$      MOVEC       MSP,D0
```

After:

```
USP=b03f8910 MSP=b03f8914 ISP =b03f8918 SR =2700 -- -- SI PL=7 -----
D0: b03f8914 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8918
PC =MOVEC$+004  NOP
```

MOVEM Instruction

Move Multiple Registers

Syntax: MOVEM <list>,<ea>
MOVEM <ea>,<list>

Description

The MOVEM instruction transfers multiple registers to or from memory. This instruction is useful for saving and restoring registers as part of a procedure call or for switching between two tasks in a multi-tasking system. MOVEM is often the fastest instruction for copying one memory area to another.

The register list consists of register names specified by slash (/) characters. You can also specify a range of registers using the minus (-) character. For instance, the list D0/D4-D6/A2 specifies registers D0,D4,D5,D6, and A2.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-			•	•		•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Word length moves from memory to registers with MOVEM always affecting all 32 bits of the registers. Each word fetched from memory is sign-extended to 32 bits before loading the register.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		
An	001	An		
(An)	010	An	•	•
(An)+	011	An	•	
-(An)	100	An		•
(d16,An)	101	An	•	•
(d8,An,Xn)	110	An	•	•
(bd,An,Xn)	110	An	•	•
([bd,An,Xn],od)	110	An	•	•
([bd,An],Xn,od)	110	An	•	•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	•
xxx.L	111	001	•	•
#<data>	111	100		
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	Dir	0	0		Size	Mode			Register		
Register Bit Mask															

Bit 10 of the first instruction word gives the transfer direction. This bit is a zero when moving from the registers to memory and one to move memory to the registers. Bit 6 of the first instruction word gives the transfer size: 0 for word transfers, 1 for longs.

The second instruction word specifies which registers are to be transferred. There is a bit for each register. The register participates in the transfer only if the bit is set. There are two forms of this bit mask: one for the predecrement addressing mode and one for all other addressing modes. These are shown below:

Predecrement Addressing Mode:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

All Other Addressing Modes:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

The registers are always stored in the order D0–D7 followed by A0–A7. D0 is at the lowest address and A7 at the highest. When all registers are stored, the memory area looks like this:

Long Data Size		Word Data Size	
Offset	Contents	Offset	Contents
$00	D0.L	$00	D0.W
$04	D1.L	$02	D1.W
$08	D2.L	$04	D2.W
$0C	D3.L	$06	D3.W
$10	D4.L	$08	D4.W
$14	D5.L	$0A	D5.W
$18	D6.L	$0C	D6.W
$1C	D7.L	$0E	D7.W
$20	A0.L	$10	A0.W
$24	A1.L	$12	A1.W
$28	A2.L	$14	A2.W
$2C	A3.L	$16	A3.W
$30	A4.L	$18	A4.W
$34	A5.L	$1A	A5.W
$38	A6.L	$1C	A6.W
$3C	A7.L	$1E	A7.W

The saved registers occupy only as much space as necessary. If, for example, register D5 had been left out of the register mask, registers D6–A7 would appear 4 or 2 bytes lower in memory.

Example The MOVEM *+$A,D0-D3/A0-A3 instruction loads the first four data registers and the first four address registers from memory. (The assembly language symbol * refers to the location of the first word of the instruction.) This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	Dir	0	0	1	Size	Mode			Register		
Register Bit Mask															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	1	1	1	1	1	0	1	0
0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

The hex value of this instruction is $4CFA 0F0F 0006. Note that the register mask precedes the PC-relative offset word. Execution yields:

Before:

```
USP=b03f88e2 MSP=b03f88e2 ISP =b03f88e2 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f88e2
PC =MOVEM$      MOVEM.L    *+$000A,D0-D3/A0-A3 ; MOVEM$+00a
```

The memory area with the new register contents has the following appearance:

```
MOVEM$+00a:   d0d0d0d0 d1d1d1d1 d2d2d2d2 d3d3d3d3
MOVEM$+01a:   a0a0a0a0 a1a1a1a1 a2a2a2a2 a3a3a3a3
```

After:

```
USP=b03f88e2 MSP=b03f88e2 ISP =b03f88e2 SR =2700 -- -- SI PL=7 -----
D0: d0d0d0d0 d1d1d1d1 d2d2d2d2 d3d3d3d3 000000d4 000000d5 000000d6 000000d7
A0: a0a0a0a0 a1a1a1a1 a2a2a2a2 a3a3a3a3 000000a4 000000a5 000000a6 b03f88e2
PC =MOVEM$+006  NOP
```

Note

All chips in the family perform an extra read bus cycle when moving from memory to registers. The location beyond the last register fetched is read erroneously. To avoid an erroneous BUSERR exception, do not perform a MOVEM from the last location in logical or physical memory.

There is an inconsistency between members of the 68000 family when using MOVEM with the predecrement and postincrement addressing modes. If the address register specified in the effective address is also in the list of registers to be loaded or stored, earlier processors store the original value (registers to memory) or load the value in memory unchanged (memory to registers). Later processors store the decremented value (registers to memory) and load the memory value plus the increment (memory to registers).

MOVEP Instruction

Move Peripheral Data

Syntax: MOVEP d(Ay),Dx
MOVEP Dx,d(Ay)

Description

The MOVEP instruction is a holdover from the 68000 and 68010. The two oldest members of the CPU family had a 16-bit data bus, and attaching 8-bit peripheral devices required programmers to access the devices using addresses incremented by 2. The MOVEP instruction is designed to assist in this process and copies 2 or 4 bytes in a data register to successive bytes whose addresses differ by 2.

The 68020 and 68030 support a feature called *dynamic bus sizing*, which makes the MOVEP instruction largely irrelevant for those chips. With dynamic bus sizing, 8-bit, 16-bit, and 32-bit devices appear the same to software (except for speed). If the two older CPUs had had this feature, MOVEP would not have been necessary. Motorola must support MOVEP forever to maintain compatibility across members of the 68000 family.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-			•	•		•	•	○	○

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

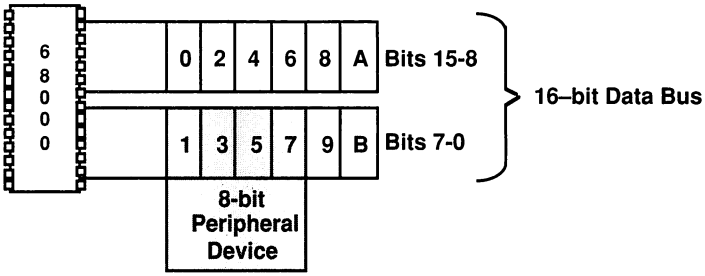
Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register Dx			1	Dir	Size	0	0	1	Register Ay		
16-bit displacement															

Bit 7 of the first instruction word gives the transfer direction. This bit is a zero when moving from memory to the data register and one to move the data register to memory. Bit 6 of the first instruction word gives the transfer size: 0 for word transfers, 1 for longs.

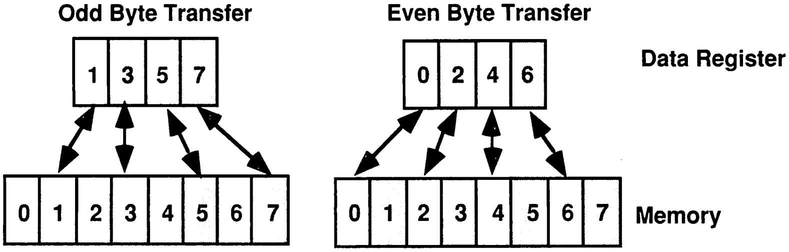
The second instruction word specifies the offset from register Ay for the first byte to be transferred. This is similar to the address register with displacement addressing mode. The instruction encoding is different, however.

MOVEP was designed for the following type of hardware configuration:


68000
0
2
4
6
8
A
Bits 15-8
1
3
5
7
9
B
Bits 7-0
16–bit Data Bus
8-bit Peripheral Device

The 68000 or 68010 has a 16-bit data bus, consisting of two 8-bit byte lanes. You can attach an 8-bit device to either the upper (bits 15–8) or lower (bits 7–0) 8-bit path. The above example shows a 4-byte device connected to the lower byte lane. MOVEP can access all four of the device's bytes with a single instruction.

If the memory address is even, transfer is made to or from the high bytes in memory. If the memory address is odd, transfer is made to or from the low bytes in memory, as shown below.


Odd Byte Transfer
1
3
5
7
0
1
2
3
4
5
6
7
Even Byte Transfer
0
2
4
6
Data Register
0
1
2
3
4
5
6
7
Memory

Example The MOVEP.L 0(A0),D0 instruction loads D0 with the even-addressed bytes from the 2 longwords indicated by A0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Register Dx			0	Dir	Size	0	0	1	Register		
16–bit displacement															

0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The hex value of this instruction is $0148 0000. Execution yields:

Before:

```
USP=b03f88c2 MSP=b03f88c2 ISP =b03f88c2 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8f98 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f88c2
PC =MOVEP$      MOVEP.L    $0000(A0),D0
```

The memory area with the new register contents has the following appearance:

```
b03f8f98:     01ff02ff 03ff04ff
```

After:

```
USP=b03f88c2 MSP=b03f88c2 ISP =b03f88c2 SR =2700 -- -- SI PL=7 -----
D0: 01020304 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8f98 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f88c2
PC =MOVEP$+004  NOP
```

MOVEQ Instruction

Move Quick

Syntax: MOVEQ #<data>,Dx

Description

The MOVEQ instruction takes advantage of the fact that many constants used in programs are small values. MOVEQ allows you to load an immediate quantity into a data register. The instruction fits into 2 bytes and allows a constant in the range -128 to +127. Use of this instruction makes programs smaller and faster.

The low-order byte of the instruction contains the immediate data. This byte is sign-extended to 32 bits and placed in the data register. The condition codes are set accordingly.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0				•		•	•	•	•

X Not affected.
N Set if result is negative (bit 31 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	Register Dx			0	8-bit immediate value							

Example The MOVEQ #1,D0 instruction loads D0 with the long constant 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	Register Dx			0	8-bit immediate value							
0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1

The hex value of this instruction is $7001. Execution yields:

Before:

```
USP=b03f88bc MSP=b03f88bc ISP =b03f88bc SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f88bc
PC =MOVEQ$       MOVEQ    #$01,D0
```

After:

```
USP=b03f88bc MSP=b03f88bc ISP =b03f88bc SR =2700 -- -- SI PL=7 -----
D0: 00000001 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f88bc
PC =MOVEQ$+002  NOP
```

MOVES Instruction (Privileged)

Move Address Space

Syntax: MOVES Rn,<ea>
MOVES <ea>,Rn

Description

The MOVES instruction allows a supervisor mode program to generate an arbitrary bus cycle. This instruction is useful for simulating accesses to coprocessor chips and other specialized hardware requiring certain types of bus cycles. MOVES can also be used in systems that use a Memory Management Unit (MMU) to move data across separate spaces managed by the MMU. For example, a multi-tasking operating system might use this instruction to transfer data to or from a user-mode task memory area during an I/O operation.

MOVES transfers a byte, word, or longword between a register and an effective address. The Source Function Code (SFC) register contains the information that will appear on the CPU FC2–FC0 bits for a transfer from the effective address to a register. The Destination Function Code contains the value for FC2–FC0 for transferring from the register to the effective address.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-		•	•	•			•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Byte and word transfers to an address register cause the data to be sign-extended during the transfer. All 32 bits of the address register are affected.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		
An	001	An		
(An)	010	An	•	•
(An)+	011	An	•	•
-(An)	100	An	•	•
(d16,An)	101	An	•	•
(d8,An,Xn)	110	An	•	•
(bd,An,Xn)	110	An	•	•
▨ ([bd,An,Xn],od)	110	An	•	•
▨ ([bd,An],Xn,od)	110	An	•	•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	•
xxx.L	111	001	•	•
#<data>	111	100		
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
▨ ([bd,PC,Xn],od)	111	011		
▨ ([bd,PC],Xn,od)	111	011		

▨ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	Size		Mode			Register		
D/A	Register Rn			Dir	0	0	0	0	0	0	0	0	0	0	0

0 = Data Register
1 = Address Register

0 = <ea> to register
1 = register to <ea>

Bits 7–6 give the size of the transfer operation: 00 for byte, 01 for word, and 10 for long operations.

Example The MOVES.L (A0),D0 instruction loads D0 with the longword whose address is in register A0. The Source Function Code Register (SFC) indicates the type of bus cycle to be run. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	Size		Mode			Register		
D/A	Register Rn			Dir	0	0	0	0	0	0	0	0	0	0	0

0	0	0	0	1	1	1	0	1	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The hex value of this instruction is $0e90 0000. Execution yields:

Before:

```
USP=b03f88a6 MSP=b03f88a6 ISP =b03f88a6 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8f48 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f88a6
PC =MOVES$      MOVES.L    (A0),D0

SFC=00000001 DFC=00000001
```

After:

```
USP=b03f88a6 MSP=b03f88a6 ISP =b03f88a6 SR =2700 -- -- SI PL=7 -----
D0: d0d0d0d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8f48 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f88a6
PC =MOVES$+004        NOP
```

The SFC register has a value of 1. Since this corresponds to the user program data space, this example loads D0 from the user mode location in register A0.

Note

The MOVES.L An,(An)+ and MOVES.L -(An),An instructions (using the same address register for both the effective address and data) yield undefined results. All current processors store or load the updated value.

MULS Instruction

Signed Multiply (Word form)

Syntax: MULS.W <ea>,Dn

Description

The MULS.W instruction multiplies a 16-bit quantity by a 16-bit quantity, yielding a 32-bit product. The effective address operand and the low word of the data register are multiplied and the result placed in all 32 bits of the data register. If the product will not fit in 32 bits, the V (overflow) condition code bit is set.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	0			•			•	•	•	•

X Not affected.
N Set if result is negative (bit 31 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An	•	
(An)+	011	An	•	
-(An)	100	An	•	
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	110	An	•	
([bd,An],Xn,od)	110	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100	•	
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Register			1	1	1	Mode			Register		

Example The MULS.W D1,D0 instruction multiplies the low-order 16 bits in D0 by the low-order 16 bits in D1. The product is placed in all 32 bits of D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Register			1	1	1	Mode			Register		
1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	1

The instruction has hex value $C1C1. Execution yields:

Before:

```
USP=b03f8852 MSP=b03f8852 ISP =b03f8852 SR =2700 -- -- SI PL=7 -----
D0: face0002 beef0003 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8852
PC =MULS$       MULS.W   D1,D0
```

After:

```
USP=b03f8852 MSP=b03f8852 ISP =b03f8852 SR =2700 -- -- SI PL=7 -----
D0: 00000006 beef0003 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8852
PC =MULS$+002   NOP
```

This example multiplies 2 (in D0) by 3 (in D1), yielding a product of 6 (in D0). Note that the high words of the two source operands are ignored.

MULS.L Instructions

Signed Multiply (Long Forms)

Syntax: MULS.L <ea>,Dl
MULS.L <ea>,Dh:Dl

Description

The 68020 and 68030 support instructions that allow 32-bit multiplication. There are two forms of 32-bit signed multiplication:

MULS.L	**<ea>,Dl**	32 bits x 32 bits yielding 32-bit result
MULS.L	**<ea>,Dh:Dl**	32 bits x 32 bits yielding 64-bit result

The effective address operand times register Dl is placed either in register Dl (32-bit result) or the concatenation of Dh and Dl (64-bit result). If the result will not fit in the destination (possible only with a 32-bit result), the V condition code bit is set.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	0				•				•	•

X	Not affected.
N	Set if result is negative (bit 63/31 = 1). Cleared otherwise.
Z	Set if result is zero. Cleared otherwise.
V	Set if an overflow occurs. Cleared otherwise.
C	Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An	•	
(An)+	011	An	•	
-(An)	100	An	•	
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	110	An	•	
([bd,An],Xn,od)	110	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100	•	
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	0	Mode			Register		
0	Register Dl			1	Size	0	0	0	0	0	0	0	Register Dh		

0 => 32–bit form. Product is in Dl
1 => 64–bit form. Product is in Dh:Dl

If bit 10 of the second instruction word (Size) is 1 and register Dh is the same as Register Dl, the results are undefined. If the Size bit is 0, the Dh field is ignored.

Example The MULS.L D2,D0:D1 instruction multiplies the longword in D1 by the longword quantity in D2. The D0 and D1 receive the high and low longwords of the result, respectively. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	0	Mode			Register		
0	Register Dl			1	Size	0	0	0	0	0	0	0	Register Dh		

0	1	0	0	1	1	0	0	0	0	0	0	0	0	1	0
0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0

The instruction has hex value $4C02 1C00. Execution yields:

Before:

```
USP=b03f8852 MSP=b03f8852 ISP =b03f8852 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 87654321 00000010 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8852
PC =MULSL$      MULS.L   D2,D0:D1
```

After:

```
USP=b03f8852 MSP=b03f8852 ISP =b03f8852 SR =2708 -- -- SI PL=7 -N---
D0: fffffff8 76543210 00000010 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8852
PC =MULSL$+004  NOP
```

This example multiplies D1 by D2, leaving the result in D0:D1. Note that the original quantity in D1 is negative, causing the upper bits of D0 and the N condition code bit to be set.

MULU Instruction

Unsigned Multiply (Word Form)

Syntax: MULU.W <ea>,Dn

Description

The MULU.W instruction multiplies a 16-bit quantity by a 16-bit quantity, yielding a 32-bit result. The multiplication is unsigned (both operands are positive). The data register always receives the product. If the product will not fit in 16 bits, the V (overflow) condition code bit is set.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	0			•			•	•	•	•

X Not affected.
N Set if result is negative (bit 31 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000	•	
An	001	An			xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100	•	
(An)+	011	An	•						
-(An)	100	An	•						
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Register			0	1	1	Mode			Register		

Example The MULU.W D1,D0 instruction multiplies the low order words of D0 and D1, placing the 32-bit result in D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	Register			0	1	1	Mode			Register		
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	1

The instruction has hex value $C0C1. Execution yields:

Before:

```
USP=b03f8852 MSP=b03f8852 ISP =b03f8852 SR =2700 -- -- SI PL=7 -----
D0: dead0002 face8000 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8852
PC =MULU$       MULU.W   D1,D0
```

After:

```
USP=b03f8852 MSP=b03f8852 ISP =b03f8852 SR =2700 -- -- SI PL=7 -----
D0: 00010000 face8000 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8852
PC =MULU$+002   NOP
```

This example multiplies hex $8000 (in D1) by 2 (in D0). Although $8000 is a negative quantity, the N bit is not set and the upper bits of the result are zero. This is due to the unsigned nature of the instruction. Note that the upper 16 bits of both source operands are ignored.

MULU.L Instructions

Unsigned Multiply (Long Forms)

Syntax: MULU.L <ea>,Dl
MULU.L <ea>,Dh:Dl

Description

The 68020 and 68030 support instructions that allow 32-bit multiplication. There are two forms of 32-bit unsigned multiplication:

MULU.L	**<ea>,Dl**	32 bits x 32 bits yielding 32-bit result
MULU.L	**<ea>,Dh:Dl**	32 bits x 32 bits yielding 64-bit result

The effective address operand times register Dl is placed either in register Dl (32-bit result) or the concatenation of Dh and Dl (64-bit result). If the result will not fit in the destination (possible only with a 32-bit result), the V condition code bit is set. All operands are treated as positive (unsigned) quantities.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	*	0				•				•	•

X Not affected.
N Set if result is negative (bit 63/31 = 1). Cleared otherwise.
Z Set if quotient is zero. Cleared otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An	•	
(An)+	011	An	•	
-(An)	100	An	•	
(d16,An)	101	An	•	
(d8,An,Xn)	110	An	•	
(bd,An,Xn)	110	An	•	
([bd,An,Xn],od)	110	An	•	
([bd,An],Xn,od)	110	An	•	

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	
xxx.L	111	001	•	
#<data>	111	100	•	
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	0	Mode			Register		
0	Register Dl			0	Size	0	0	0	0	0	0	0	Register Dh		

0 => 32–bit form. Product is in Dl
1 => 64–bit form. Product is in Dh:Dl

If bit 10 of the second instruction word (Size) is 1 and register Dh is the same as Register Dl, the results are undefined. If the Size bit is 0, the Dh field is ignored.

Example The MULU.L D2,D0:D1 instruction multiplies D2 and D1, leaving the result in D0 and D1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	0	Mode			Register		
0	Register Dl			0	Size	0	0	0	0	0	0	0	Register Dh		

0	1	0	0	1	1	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0

The instruction has hex value $4C02 1400. Execution yields:

Before:

```
USP=b03f8852 MSP=b03f8852 ISP =b03f8852 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 87654321 00000010 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8852
PC =MULUL$      MULU.L   D2,D0:D1
```

After:

```
USP=b03f8852 MSP=b03f8852 ISP =b03f8852 SR =2700 -- -- SI PL=7 -----
D0: 00000008 76543210 00000010 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8852
PC =MULUL$+004  NOP
```

Note that the original quantity in D1 is negative, but that the upper bits of the result (in D0) are zero. The N bit also remained clear, indicating the unsigned nature of the instruction.

NBCD Instruction

Negate Decimal With Extend

Syntax: NBCD <ea>

Description

The NBCD instruction subtracts the destination operand and X condition code bit from zero. You can use this instruction to invert the sign of a BCD number that is possibly a multiple-precision number. NBCD works only on a byte operand, however. Negating a BCD number effectively subtracts that number from 100 (decimal).

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	U	*	U	*		•				•	•	•	•

X Set if a decimal borrow occurs. Cleared otherwise.
N Undefined.
Z Cleared if result is non-zero. Unchanged otherwise.
V Undefined.
C Set if a decimal borrow occurs. Cleared otherwise.

You must ensure that the X bit (decimal carry) is in the appropriate state for this instruction. Before a single-byte operation or the first in a series of multiple-byte operations, the bit must be cleared. The second and subsequent operations in a multiple-byte operation use the X bit setting generated by the previous step.

The behavior of the Z condition code follows the 68000 standard for multiple-precision arithmetic. You normally set this bit before beginning a multiple-precision operation. If it is still set at the end of the operation, the entire result is zero.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100		
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	Mode			Register		

Example The NBCD D0 instruction negates the BCD number in the low-order byte of D0. This instruction has the following form:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	Mode			Register		
0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0

The hex value of the instruction is $4800. Execution yields:

Before:

```
USP=b03f8842 MSP=b03f8842 ISP =b03f8842 SR =2700 -- -- SI PL=7 -----
D0: face0f99 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8842
PC =NBCD$       NBCD       D0
```

After:

```
USP=b03f8842 MSP=b03f8842 ISP =b03f8842 SR =2711 -- -- SI PL=7 X---C
D0: face0f01 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8842
PC =NBCD$+002   NOP
```

This example takes the negative of decimal 99, producing 01. A decimal borrow occurred, setting both the N and C condition code bits. Note that the upper 24 bits of the register were not affected.

NEG Instruction

Negate Binary

Syntax: NEG <ea>

Description

The NEG instruction subtracts the destination operand from zero. You can use this instruction to invert the sign of a two's complement binary number.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•	•	•		•	•	•	•

X Cleared if the result is zero. Set otherwise.
N Set if the result is negative. Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Cleared if the result is zero. Set otherwise.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100		
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	Size		Mode			Register		

00=Byte
01=Word
10=Long

Example The NEG.L D0 instruction negates the binary number in D0. This instruction has the following form:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	Size		Mode			Register		
0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0

The hex value of the instruction is $4480. Execution yields:

Before:

```
USP=b03f8836 MSP=b03f8836 ISP =b03f8836 SR =2700 -- -- SI PL=7 -----
D0: 00000001 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8836
PC =NEG$       NEG.L     D0
```

After:

```
USP=b03f8836 MSP=b03f8836 ISP =b03f8836 SR =2719 -- -- SI PL=7 XN--C
D0: ffffffff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8836
PC =NEG$+002   NOP
```

This example negates positive 1 to become negative 1. Note that the result is non-zero and negative, thus setting the X, N, and C condition code bits.

NEGX Instruction

Negate Binary with Extend

Syntax: NEGX <ea>

Description

The NEGX instruction subtracts the destination operand and the extend bit from zero. You can use this instruction to invert the sign of a 2's complement binary number that is possibly a multiple-precision number.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•	•	•		•	•	•	•

X Cleared if the result is zero. Set otherwise.
N Set if the result is negative. Cleared otherwise.
Z Cleared if the result is non-zero. Unchanged otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Cleared if the result is zero. Set otherwise.

The behavior of the Z condition code follows the 68000 standard for multiple-precision arithmetic. You normally set this bit before beginning a multiple-precision operation. If it is still set at the end of the operation, the entire result is zero.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•
An	001	An		
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100		
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	Size		Mode			Register		

00=Byte
01=Word
10=Long

Example The NEGX.L D0 instruction negates the binary number in D0, with the X condition code bit participating. This instruction has the following form:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	Size		Mode			Register		
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0

The hex value of the instruction is $4080. The example illustrates the technique for negating a multiple precision number. Here we negate the 64-bit quantity in D0:D1. You first perform a NEG.L instruction on the low-order 32 bits (in D1) and then perform a NEGX.L on the high-order part (in D0). More than 64 bits requires additional NEGX operations. Execution yields:

Before:

```
USP=b03f8826 MSP=b03f8826 ISP =b03f8826 SR =2700 -- -- SI PL=7 -----
D0: 00000000 00000001 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8826
PC =NEGX$       NEG.L    D1

USP=b03f8826 MSP=b03f8826 ISP =b03f8826 SR =2719 -- -- SI PL=7 XN--C
D0: 00000000 ffffffff 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8826
PC =NEGX$+002 NEGX.L    D0
```

After:

```
USP=b03f8826 MSP=b03f8826 ISP =b03f8826 SR =2719 -- -- SI PL=7 XN--C
D0: ffffffff ffffffff 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8826
PC =NEGX$+004   NOP
```

This example negates positive 1 to become negative 1 (as a 64-bit quantity). The result is non-zero and negative, thus setting the X, N, and C condition code bits. Note the carry between the two parts of the number through the X bit.

NOP Instruction

No Operation

Syntax: NOP

Description

The NOP instruction performs no action. NOP instructions are useful for adding memory space or small amounts of time in a program. External hardware frequently cannot respond quickly enough to successive accesses by the 68000. Thus you must often use one or more NOP instructions between accesses to this type of hardware to ensure proper operation.

The 68020 and 68030 complicate this issue due to instruction pipelining and memory cache operation. Since these chips have an instruction pipeline, the instruction execution and the corresponding memory bus cycles are not always synchronized. On the 68020 and 68030, all bus cycles are completed before the NOP instruction begins execution, thus preserving the ability to insert time delays into an instruction stream. Note that improper operation can still result on the 68030 due to cache burst mode. You may have to explicitly disable burst mode when access timing is critical.

Another interesting programming trick when using a non-symbolic debugger with memory search capability is to place a string of NOP instructions in front of a procedure you wish to locate. For instance, if you place a string of two NOPs in front of procedure A, and three NOPs in front of procedure B, you can locate these procedures without symbols. Simply search for three NOPs (B), then two NOPs (A and B).

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-						•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1

The value of a NOP instruction is hex $4E71.

Example A NOP instruction advances the PC by two bytes, as shown below:

Before:

```
USP=b03f8810 MSP=b03f8810 ISP =b03f8810 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8810
PC =NOP$        NOP
```

After:

```
USP=b03f8810 MSP=b03f8810 ISP =b03f8810 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8810
PC =NOP$+002    NOP
```

NOT Instruction

One's Complement

Syntax: NOT <ea>

Description

The NOT instruction simply inverts all bits in the destination. This operation is also known as taking the *one's complement.*

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0		•	•	•		•	•	•	•

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100		
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	Size		Mode			Register		

00=Byte
01=Word
10=Long

Example The NOT.L D0 instruction inverts all bits in D0. This instruction has the following form:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	Size		Mode			Register		
0	1	0	0	0	1	1	0	1	0	0	0	0	0	0	0

The hex value of the instruction is $4680. Execution yields:

Before:

```
USP=b03f8810 MSP=b03f8810 ISP =b03f8810 SR =2700 -- -- SI PL=7 -----
D0: 76543210 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8810
PC =NOT$        NOT.L      D0
```

After:

```
USP=b03f8810 MSP=b03f8810 ISP =b03f8810 SR =2708 -- -- SI PL=7 -N---
D0: 89abcdef 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8810
PC =NOT$+002    NOP
```

OR Instruction

Inclusive OR Logical

Syntax: OR Dn,<ea>
OR <ea>,Dn

Description

The OR instruction performs an inclusive OR operation between a data register and an effective address operand.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0		•	•	•		•	•	•	•

X Not affected.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Valid Addressing Modes

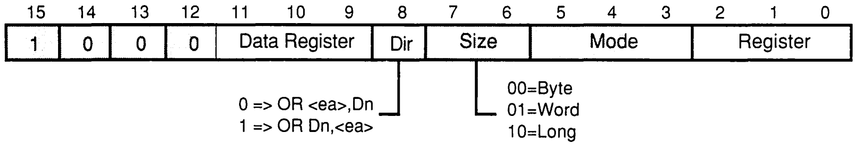
Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	
An	001	An		
(An)	010	An	•	•
(An)+	011	An	•	•
-(An)	100	An	•	•
(d16,An)	101	An	•	•
(d8,An,Xn)	110	An	•	•
(bd,An,Xn)	110	An	•	•
([bd,An,Xn],od)	110	An	•	•
([bd,An],Xn,od)	110	An	•	•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	•
xxx.L	111	001	•	•
#<data>	111	100	•	
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

If you specify an immediate mode source, many assemblers will generate an ORI instruction. A data register destination must use the OR <ea>,Dn form of the instruction (see below).

Instruction Format



Example The OR.L D0,D1 instruction performs a 32-bit logical OR between the contents of D0 and the contents of D1. The result is placed in D1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Data Register			DIR	Size		Mode			Register		
1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0

The hex value of the instruction is $8280. Instruction execution yields:

Before:

```
USP=b03f87ce MSP=b03f87ce ISP =b03f87ce SR =2700 -- -- SI PL=7 -----
D0: 76543210 89abcdef 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f87ce
PC =OR$        OR.L      D0,D1
```

After:

```
USP=b03f87ce MSP=b03f87ce ISP =b03f87ce SR =2708 -- -- SI PL=7 -N---
D0: 76543210 ffffffff 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f87ce
PC =OR$+002    NOP
```

ORI Instruction

Inclusive OR Immediate

Syntax: ORI #<data>,<ea>

Description

The ORI instruction performs an inclusive OR operation on an effective address operand using immediate data.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0		•	•	•		•	•	•	•

X Not affected.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100	•	
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	Size		Mode			Register		
Word Data (16 bits)								Byte Data (8 bits)							
Long Data															

The size field indicates the operation size: 00 for byte, 01 for word, and 10 for long. There are 2 or 4 bytes of immediate data that follow the instruction in memory. Byte and word operations have 16 bits (2 bytes) of immediate data. Long operations have 32 bits (4 bytes).

Example The instruction ORI.L #$89ABCDEF,D0 performs an Exclusive OR between the 32-bit value in register D0 with the hex value $89ABCDEF. The immediate data is contained in the two words following the instruction, as shown below:

Instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	Size		Mode			Register		
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Data

1	0	0	0	1	0	0	1	1	0	1	0	1	0	1	1
1	1	0	0	1	1	0	1	1	1	1	0	1	1	1	1

This instruction has hex value $0080 89AB CDEF. Executing the instruction yields:

Before:

```
USP=b03f87ce MSP=b03f87ce ISP =b03f87ce SR =2700 -- -- SI PL=7 -----
D0: 76543210 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f87ce
PC =ORI$       ORI.L      #$89ABCDEF,D0
```

After:

```
USP=b03f87ce MSP=b03f87ce ISP =b03f87ce SR =2708 -- -- SI PL=7 -N---
D0: ffffffff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f87ce
PC =ORI$+006    NOP
```

ORI to CCR Instruction

Syntax: ORI #<data>,CCR

Inclusive OR Immediate to Condition Code Register

Description

The ORI to CCR instruction performs a logical Inclusive OR operation on the Condition Code Register using an 8-bit immediate field. This is useful for setting any arbitrary combination of condition code bits.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•				•	•	•	•

X	Set if bit 4 of immediate operand is one. Unchanged otherwise.
N	Set if bit 3 of immediate operand is one. Unchanged otherwise.
Z	Set if bit 2 of immediate operand is one. Unchanged otherwise.
V	Set if bit 1 of immediate operand is one. Unchanged otherwise.
C	Set if bit 0 of immediate operand is one. Unchanged otherwise.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	Byte Data (8 bits)							

Example The ORI #$06,CCR instruction sets the Z and V condition code bits. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

The instruction has hex value $003C 0006. Execution yields:

Before:

```
USP=b03f87ce MSP=b03f87ce ISP =b03f87ce SR =2719 -- -- SI PL=7 XN--C
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f87ce
PC =ORICCR$     ORI.B    #$06,CCR
```

After:

```
USP=b03f87ce MSP=b03f87ce ISP =b03f87ce SR =271f -- -- SI PL=7 XNZVC
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f87ce
PC =ORICCR$+004          NOP
```

ORI to SR Instruction (Privileged) Syntax: ORI #<data>,SR

Inclusive OR Immediate to Status Register

Description

The ORI to SR instruction performs an Inclusive OR operation on the Status Register using a 16-bit immediate field. This is useful for setting any arbitrary combination of bits in this register. This processor must be in Supervisor mode to execute this instruction. Attempted execution in user mode will cause a Privilege Violation exception through vector 8.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*			•			•	•	•	•

X Set if bit 4 of immediate operand is one. Unchanged otherwise.
N Set if bit 3 of immediate operand is one. Unchanged otherwise.
Z Set if bit 2 of immediate operand is one. Unchanged otherwise.
V Set if bit 1 of immediate operand is one. Unchanged otherwise.
C Set if bit 0 of immediate operand is one. Unchanged otherwise.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0
Word Data (16 bits)															

Example The ORI #$303,SR instruction sets the V and C condition code bits, as well as the low two bits of the interrupt mask. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0
0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1

The instruction has hex value $007C 0303. Execution yields:

Before:

```
USP=b03f87ce MSP=b03f87ce ISP =b03f87ce SR =241c -- -- SI PL=4 XNZ--
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f87ce
PC =ORISR$      ORI.W    #$0303,SR
```

After:

```
USP=b03f87ce MSP=b03f87ce ISP =b03f87ce SR =271f -- -- SI PL=7 XNZVC
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f87ce
PC =ORISR$+004  NOP
```

PACK Instruction

Pack BCD

Syntax: PACK -(Ax),-(Ay),#<data>
PACK Dx,Dy,#<data>

Description

The PACK instruction converts two characters into a single-byte BCD quantity. This instruction is normally used to convert ASCII or EBCDIC digits into BCD. ASCII digits are in the range hex $30–$39 (0–9), and EBCDIC digits are in the range hex $F0–$F9 (0–9). The American Standard Code for Information Interchange (ASCII) is the dominant standard for characters currently. The older IBM EBCDIC (Extended Binary Coded Decimal Interchange Code) is widely used in IBM mainframe computers.

The PACK instruction takes a word quantity, adds an immediate value, and concatenates the low order 4 bits (called a nibble) of both bytes into a single byte. The immediate value allows the use of a character set other than ASCII or EBCDIC. For the two standard character sets, the immediate value should be zero.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-		•						•	•

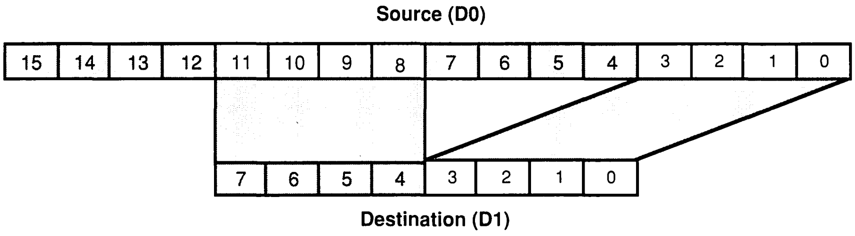
X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Register Rx			1	0	1	0	0	R/M	Register Ry		
16-bit immediate value															

Bit 3 (R/M) of the first instruction word is zero for the data register form of the instruction, and one for the predecrement addressing mode form.

Example The PACK D0,D1,#0 instruction places bits 11–8 and 3–0 of register D0 in the low byte of D1, as shown below:



This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Register Rx			1	0	1	0	0	R/M	Register Ry		
16-bit immediate value															

1	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The instruction has hex value $8340 0000. Execution yields:

Before:

```
USP=b03f87ba MSP=b03f87ba ISP =b03f87ba SR =2700 -- -- SI PL=7 -----
D0: 00003132 d1d1d1d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f87ba
PC =PACK$       PACK        D0,D1,#$0000
```

After:

```
USP=b03f87ba MSP=b03f87ba ISP =b03f87ba SR =2700 -- -- SI PL=7 -----
D0: 00003132 d1d1d112 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f87ba
PC =PACK$+004   NOP
```

This example converts ASCII 12 to BCD 12. The high-order 24 bits of the destination are undisturbed.

PEA Instruction

Syntax: PEA <ea>

Push Effective Address

Description

The PEA instruction evaluates an effective address and pushes this address onto the stack. The address is always a 32-bit quantity.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-				•		•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn			xxx.W	111	000	•	
An	001	An			xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100		
(An)+	011	An							
-(An)	100	An							
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	Mode			Register		

Example The PEA 4(A0) instruction pushes the contents of register A0 plus 4 onto the stack. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	Mode			Register		
0	1	0	0	1	0	0	0	0	1	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

The hex value of this instruction is $4868 0004. Execution yields:

Before:

```
USP=b03f87ae MSP=b03f87ae ISP =b03f87ae SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 00000008 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f87ae
PC =PEA$        PEA      $0004(A0)
```

After:

```
USP=b03f87ae MSP=b03f87ae ISP =b03f87aa SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 00000008 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f87aa
PC =PEA$+004    NOP
```

The value stored in memory is shown below:

```
b03f87aa:     0000000c
```

RESET Instruction (Privileged)

Syntax: **RESET**

Reset External Devices

Description

The RESET instruction asserts the RESET signal for 512 CPU clock cycles. This action causes all external devices to return to a reset state. The processor itself is unaffected, except for the time required by the instruction and the possible need to re-initialize some external hardware.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-						•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0

The RESET instruction has hex value $4E70.

ROL Instruction

Rotate Left (Without Extend)

Syntax:
ROL Dx,Dy
ROL #<data>,Dy
ROL <ea>

Description

The ROL instruction performs a logical rotate left on a data register or on an effective address operand. The shift count for a data register can be specified one of two ways: The low 5 bits of a data register can be used for the count value (ROL Dx,Dy form), or a small constant in the instruction itself can hold the count (ROL #<data>,Dy form). The register form allows shifts of up to 63 bits, while the immediate constant can have a value in the range 1–8. The data register form of the instruction supports all sizes: byte, word, and longword.

The effective address form shifts only 1 bit and supports only the word data size.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	*		•	•	•		•	•	•	•

☐ Register form only

X	Not affected.
N	Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z	Set if result is zero. Cleared otherwise.
V	Always cleared.
C	Set to the last bit shifted out of the operand. Cleared for a rotate count of zero.

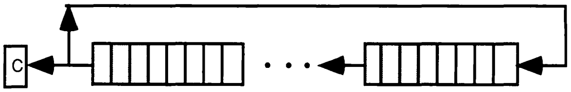
Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100	•	
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

☐ 68020/030 only ☐ Register form only

Bits shifted out of the most significant bit are placed in both the Carry (C) condition code bit and the least significant bit of the operand, as shown below:



Instruction Format

Register Shifts

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			1	Size		i/r	1	1	Register		

00=Byte
01=Word
10=Long

0 = Immediate Shift count
1 = Register Shift count

The Count/Register field contains either a register number (bit 5 = 1) or an immediate shift count (bit 5 = 0). The immediate shift count form uses bits 11–9 to hold a value from 1–7. A value of zero in these bits represents a value of 8. Bits 2–0 contain the number of the register to be shifted.

Memory Shifts

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	1	1	1	Mode			Register		

Bit 8 in both formats indicates the shift direction. A one indicates a left shift, and zero indicates a right shift.

Example The ROL.L D1,D0 instruction rotates the 32-bit quantity in D0 left by the number of bits specified by register D1. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			1	Size		i/r	1	1	Register		
1	1	1	0	0	0	1	1	1	0	1	1	1	0	0	0

The instruction has hex value $E3B8. Execution yields:

Before:

```
USP=b03f8766 MSP=b03f8766 ISP =b03f8762 SR =2700 -- -- SI PL=7 -----
D0: 87654321 00000010 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8762
PC =ROL$         ROL.L    D1,D0
```

After:

```
USP=b03f8766 MSP=b03f8766 ISP =b03f8762 SR =2701 -- -- SI PL=7 ----C
D0: 43218765 00000010 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8762
PC =ROL$+002    NOP
```

In this example, the shift count is $10 (16 decimal). This instruction swaps the words in D0.

ROR Instruction

Rotate Right (Without Extend)

Syntax: ROR Dx,Dy
ROR #<data>,Dy
ROR <ea>

Description

The ROR instruction performs a logical rotate right on a data register or on an effective address operand. The shift count for a data register can be specified one of two ways: The low 5 bits of a data register can be used for the count value (ROR Dx,Dy form), or a small constant in the instruction itself can hold the count (ROR #<data>,Dy form). The register form allows shifts of up to 63 bits, and the immediate constant can have a value in the range 1–8. The data register form of the instruction supports all sizes: byte, word, and longword.

The effective address form shifts only 1 bit and supports only the word data size.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	*		•	•	•		•	•	•	•

☐ Register form only

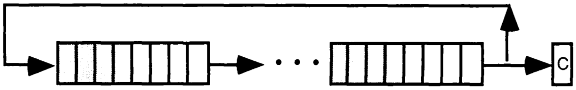
X Unaffected.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Set to the last bit shifted out of the operand. Cleared for a rotate count of zero.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000		•
An	001	An			xxx.L	111	001		•
(An)	010	An		•	#<data>	111	100	•	
(An)+	011	An		•					
-(An)	100	An		•					
(d16,An)	101	An		•	(d16,PC)	111	010		
(d8,An,Xn)	110	An		•	(d8,PC,Xn)	111	011		
(bd,An,Xn)	110	An		•	(bd,PC,Xn)	111	011		
([bd,An,Xn],od)	110	An		•	([bd,PC,Xn],od)	111	011		
([bd,An],Xn,od)	110	An		•	([bd,PC],Xn,od)	111	011		

☐ 68020/030 only ☐ Register form only

Bits shifted out of the least significant bit are placed in both the Carry (C) condition code bit and the most significant bit of the operand, as shown below:



Instruction Format

Register Rotates

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			0	Size		i/r	1	1	Register		

00=Byte
01=Word
10=Long

0 = Immediate Shift count
1 = Register Shift count

The Count/Register field contains either a register number (bit 5 = 1) or an immediate shift count (bit 5 = 0). The immediate shift count form uses bits 11–9 to hold a value from 1–7. A value of zero in these bits represents a value of 8. Bits 2–0 contain the number of the register to be shifted.

Memory Rotates

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	0	1	1	Mode			Register		

Bit 8 in both formats indicates the shift direction. A one indicates a left shift, and zero indicates a right shift.

Example The ROR.L D1,D0 instruction rotates the 32-bit quantity in D0 right by the number of bits specified by register D1. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			0	Size		i/r	1	1	Register		
1	1	1	0	0	0	1	0	1	0	1	1	1	0	0	0

The instruction has hex value $E2B8. Execution yields:

Before:
```
USP=b03f8766 MSP=b03f8766 ISP =b03f8762 SR =2700 -- -- SI PL=7 -----
D0: 87654321 00000010 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8762
PC =ROR$        ROR.L    D1,D0
```

After:
```
USP=b03f8766 MSP=b03f8766 ISP =b03f8762 SR =2700 -- -- SI PL=7 -----
D0: 43218765 00000010 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8762
PC =ROR$+002    NOP
```

In this example, the shift count is $10 (16 decimal). This instruction swaps the words in D0.

ROXL Instruction

Rotate Left (With Extend)

Syntax: ROXL Dx,Dy
ROXL #<data>,Dy
ROXL <ea>

Description

The ROXL instruction performs a logical rotate left (with extend) on a data register or on an effective address operand. The shift count for a data register can be specified one of two ways: The low 5 bits of a data register can be used for the count value (ROXL Dx,Dy form), or a small constant in the instruction itself can hold the count (ROXL #<data>,Dy form). The register form allows shifts of up to 63 bits, and the immediate constant can have a value in the range 1–8. The data register form of the instruction supports all sizes: byte, word, and longword.

The effective address form shifts only 1 bit and supports only the word data size.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	*		•	•	•		•	•	•	•

☐ Register form only

X Set according to the last bit rotated out of the operand. Unaffected when rotate count is zero.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Set to the last bit shifted out of the operand. Set to value of the X bit for a rotate count of zero.

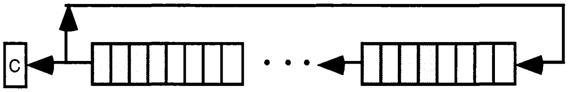
Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100	•	
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

☐ 68020/030 only ☐ Register form only

Bits shifted out of the most significant bit are placed in both the Carry (C) and eXtend (X) condition code bits. The least significant bit of the operand is set to the old value of the X bit, as shown below:



Instruction Format

Register Rotates

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			1	Size		i/r	1	0	Register		

00=Byte
01=Word
10=Long

0 = Immediate Shift count
1 = Register Shift count

The Count/Register field contains either a register number (bit 5 = 1) or an immediate shift count (bit 5 = 0). The immediate shift count form uses bits 11–9 to hold a value from 1–7. A value of zero in these bits represents a value of 8. Bits 2–0 contain the number of the register to be shifted.

Memory Rotates

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	1	1	1	Mode			Register		

Bit 8 in both formats indicates the shift direction. A one indicates a left shift, and zero indicates a right shift.

Example The ROXL.L D1,D0 instruction rotates the 32-bit quantity in D0 left by the number of bits specified by register D1. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			1	Size		i/r	1	0	Register		
1	1	1	0	0	0	1	1	1	0	1	1	0	0	0	0

The instruction has hex value $E3B0. Execution yields:

Before:

```
USP=b03f8766 MSP=b03f8766 ISP =b03f8762 SR =2700 -- -- SI PL=7 -----
D0: 87654321 00000010 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8762
PC =ROXL$       ROXL.L     D1,D0
```

After:

```
USP=b03f8766 MSP=b03f8766 ISP =b03f8762 SR =2711 -- -- SI PL=7 X---C
D0: 432143b2 00000010 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8762
PC =ROXL$+002   NOP
```

In this example, the shift count is $10 (16 decimal).

ROXR Instruction
Rotate Right (With Extend)

Syntax: ROXR Dx,Dy
ROXR #<data>,Dy
ROXR <ea>

Description

The ROXR instruction performs a logical rotate right (with extend) on a data register or on an effective address operand. The shift count for a data register can be specified one of two ways: The low 5 bits of a data register can be used for the count value (ROXR Dx,Dy form), or a small constant in the instruction itself can hold the count (ROXR #<data>,Dy form). The register form allows shifts of up to 63 bits, and the immediate constant can have a value in the range 1–8. The data register form of the instruction supports all sizes: byte, word, and longword.

The effective address form shifts only 1 bit and supports only the word data size.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	*		•	•	•		•	•	•	•

☐ Register form only

X Set according to the last bit rotated out of the operand. Unaffected when rotate count is zero.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Always cleared.
C Set to the last bit shifted out of the operand. Set to value of the X bit for a rotate count of zero.

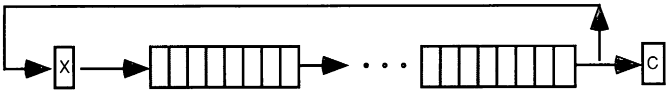
Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•
An	001	An		
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100	•	
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

☐ 68020/030 only ☐ Register form only

Bits shifted out of the least significant bit are placed in both the Carry (C) and eXtend condition code bits. The old value of the X bit is transferred to the most significant bit of the operand, as shown below:



Instruction Format

Register Rotates

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			0	Size		i/r	1	0	Register		

00=Byte
01=Word
10=Long

0 = Immediate Shift count
1 = Register Shift count

The Count/Register field contains either a register number (bit 5 = 1) or an immediate shift count (bit 5 = 0). The immediate shift count form uses bits 11–9 to hold a value from 1–7. A value of zero in these bits represents a value of 8. Bits 2–0 contain the number of the register to be shifted.

Memory Rotates

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	1	1	Mode			Register		

Bit 8 in both formats indicates the shift direction. A one indicates a left shift, and zero indicates a right shift.

Example The ROXR.L D1,D0 instruction rotates the 32-bit quantity in D0 right by the number of bits specified by register D1. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	Count/Register			0	Size		i/r	1	0	Register		
1	1	1	0	0	0	1	0	1	0	1	1	0	0	0	0

The instruction has hex value $E2B0. Execution yields:

Before:

```
USP=b03f8766 MSP=b03f8766 ISP =b03f8762 SR =2700 -- -- SI PL=7 -----
D0: 87654321 00000010 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8762
PC =ROXR$       ROXR.L     D1,D0
```

After:

```
USP=b03f8766 MSP=b03f8766 ISP =b03f8762 SR =2708 -- -- SI PL=7 -N---
D0: 86428765 00000010 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8762
PC =ROXR$+002   NOP
```

In this example, the shift count is $10 (16 decimal).

RTD Instruction

Return and Deallocate

Syntax: RTD #<data>

Description

The RTD instruction pops the longword on the top of the stack into the program counter and adds a sign-extended 16-bit value to the stack pointer. The value in the stack pointer at the end of the instruction is the original contents plus the immediate data plus 4. This instruction is intended for use in returning from a subroutine. Many high level languages call subroutines by pushing the arguments on the stack and executing a BSR or JSR instruction. The RTD instruction allows returning from the subroutine and popping off the arguments in a single instruction.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-							•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	0
16–bit Displacement															

Example The RTD #8 instruction transfers control to the location contained in the longword at the top of the stack and increments the stack pointer (A7) by 12 (hex $C). This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

The hex value of this instruction is $4E74 0008. Execution yields:

Before:

```
USP=b03f870a MSP=b03f870a ISP =b03f86fa SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f86fa
PC =RTD$        RTD        #$0008
```

The stack contains the following values:

```
b03f86fa:     b03f8f6a 01234567 89abcdef
```

After:

```
USP=b03f870a MSP=b03f870a ISP =b03f8706 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8706
PC =RTD$EXIT    NOP
```

Only register A7 (the stack pointer) and the PC change as a result of this instruction.

RTE Instruction (Privileged) Syntax: RTE

Return from Exception

Description

The RTE instruction pops the word on the top of the stack into the status register and the next longword on the stack into the program counter. RTE allows resuming an instruction stream suspended by an interrupt or program fault.

The 68000 uses the 6-byte stack frame described above. The 68010, 68020, and 68030 have a 4-word (8-byte) stack frame, with the fourth word containing a format code in the upper 4 bits. There may be additional information in the stack frame, as described below.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*						•	•	•	•

X Loaded from bit 4 of the word on top of the stack before instruction execution.
N Loaded from bit 3 of the word on top of the stack before instruction execution.
Z Loaded from bit 2 of the word on top of the stack before instruction execution.
V Loaded from bit 1 of the word on top of the stack before instruction execution.
C Loaded from bit 0 of the word on top of the stack before instruction execution.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1

The RTE instruction always has the hex value $4E73.

Example Execution yields:

Before:

```
USP=b03f870a MSP=b03f870a ISP =b03f8700 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8700
PC =RTE$       RTE
```

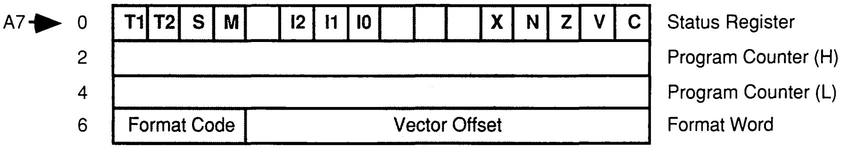
The stack contains the following values:

```
b03f8700: 241f b03f 8f7e 0000
```

After:

```
USP=b03f870a MSP=b03f870a ISP =b03f8708 SR =241f -- -- SI PL=4 XNZVC
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8708
PC =RTE$EXIT    NOP
```

Stack Frames The 68010 and later CPU chips use a different format than the 68000 for the RTE instruction. The 68000 RTE instruction uses a 6-byte stack frame consisting of the new status register contents (16 bits) and the new program counter (32 bits). The newer chips have an additional word following the program counter contents called the *format word.* The upper 4 bits of this word give an indication of the stack frame format. For some formats, additional information appears beyond the format word, as shown below:



(Optional Additional Information)

The format code (bits 15–12 of the format word) can have 16 possible values. These format codes are explained fully in Chapter 2. The codes are summarized below:

Format	**Processors**	**Frame Size**	**Purpose**
0	68010/20/30	8	Default stack frame
1	68020/30	8	Throwaway stack frame (switches to master stack)
2	68020/30	12	Instruction error (illegal instruction, Line A, etc.)
8	68010	58	BUSERR and addressing error (long form)
9	68020/30	20	Coprocessor mid instruction exception
10	68020/30	32	BUSERR and addressing error (short form)
11	68020/30	92	BUSERR and addressing error (long form)

A value which is not valid for a particular processor will cause that processor to take a format error exception.

RTM Instruction

Syntax: RTM Rn

Return from Module

Description

The RTM instruction restores a module from the stack. After the module state is restored, the caller's stack pointer is incremented by the argument count value in the module state. A description of module states appears in the CALLM instruction Description. This instruction is supported only by the 68020.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*								•	

X Loaded from bit 4 of the word at offset 2 in the stack frame.
N Loaded from bit 3 of the word at offset 2 in the stack frame.
Z Loaded from bit 2 of the word at offset 2 in the stack frame.
V Loaded from bit 1 of the word at offset 2 in the stack frame.
C Loaded from bit 0 of the word at offset 2 in the stack frame.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	0	D/A	Register Rn		

The D/A field is zero if the module data pointer (register Rn) is a data register and one if an address register.

Example The RTM A0 instruction restores the module state on top of the stack. A0 is the module data pointer restored from offset $10 of the stack frame. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	0	0	D/A	Register Rn		
0	0	0	0	0	1	1	0	1	1	0	0	1	0	0	0

This instruction has hex value $06C8. Execution yields:

Before:

```
USP=b03f85ac MSP=b03f85ac ISP =b03f858c SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: a0a0a0a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f858c
PC =RTM$        RTM        A0
```

The stack has the following layout:

```
b03f858c:     80000000 000800a5 b03f89ee b03f89e4
b03f859c:     000000a0 b03f85a4 fedcba98 76543210
```

The return will be to address $B03F89E4, which is the instruction following the CALLM instruction at label CALLM$+6. The other data areas are explained in the example for the CALLM instruction.

After:

```
USP=b03f85ac MSP=b03f85ac ISP =b03f85a4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f85a4
PC =CALLM$+006  NOP
```

RTR Instruction

Syntax: RTR

Return and Restore Condition Codes

Description

The RTR instruction pops the word on the top of the stack into the condition code register and the next longword on the stack into the program counter. The system byte of the status register is not affected. RTR provides user mode programs with many of the capabilities found in the RTE instruction, in a non-privileged environment.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*						•	•	•	•

X Loaded from bit 4 of the word on top of the stack before instruction execution.
N Loaded from bit 3 of the word on top of the stack before instruction execution.
Z Loaded from bit 2 of the word on top of the stack before instruction execution.
V Loaded from bit 1 of the word on top of the stack before instruction execution.
C Loaded from bit 0 of the word on top of the stack before instruction execution.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	1

The RTR instruction always has the hex value $4E77.

Example Execution yields:

Before:

```
USP=b03f870a MSP=b03f870a ISP =b03f8702 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8702
PC =RTR$        RTR
```

The stack contains the following values:

```
b03f8702:001f b03f 8f94
```

After:

```
USP=b03f870a MSP=b03f870a ISP =b03f8708 SR =271f -- -- SI PL=7 XNZVC
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8708
PC =RTR$EXIT    NOP
```

RTS Instruction

Syntax: **RTS**

Return from Subroutine

Description

The RTS instruction pops the longword on the top of the stack into the program counter. This instruction is normally used to reverse the action of a BSR (Branch to Subroutine) or JSR (Jump to Subroutine) instruction.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-						•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1

The RTS instruction always has the hex value $4E75.

Example Execution yields:

Before:

```
USP=b03f870a MSP=b03f870a ISP =b03f8704 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8704
PC =RTS$        RTS
```

After:

```
USP=b03f870a MSP=b03f870a ISP =b03f8708 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8708
PC =RTS$EXIT    NOP
```

SBCD Instruction

Syntax: SBCD Dy,Dx
SBCD -(Ay),-(Ax)

Subtract Binary-Coded Decimal with Extend

Description

The SBCD instruction subtracts two 8-bit binary-coded decimal (BCD) numbers to produce an 8-bit result. The eXtend condition code bit participates in the subtraction and is set by the result. The instruction can have two forms:

1. Subtract a data register from another data register.
2. Subtract two bytes in memory using the Address Register Predecrement addressing mode.

The 68000 big-endian data format dictates the form of multiple-precision operations with the SBCD instruction. The memory-to-memory form must begin at the highest-addressed byte and work back to the lowest-addressed byte, using predecrement addressing.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	U	*	U	*		•				•	•	•	•

X Set if decimal borrow was generated. Cleared otherwise.
N Undefined.
Z Cleared if the result is non-zero. Unchanged otherwise.
V Undefined.
C Set if decimal borrow was generated. Cleared otherwise.

You must ensure that the X bit (decimal carry) is in the appropriate state for this instruction. Before a single-byte operation or the first in a series of multiple-byte operations, the bit must be cleared. The second and subsequent operations in a multiple-byte operation use the X bit setting generated by the previous step.

The behavior of the Z condition code follows the 68000 standard for multiple-precision arithmetic. You normally set this bit before beginning a multiple-precision operation. If it is still set at the end of the operation, the entire result is zero.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Register Rx			1	0	0	0	0	R/M	Register Ry		

Bits 11–9 and 2–0 contain the two register numbers indicated in the syntax description. Bit 3 (R/M) indicates whether the instruction is a register-to-register operation (R/M = 0) or a memory-to-memory operation (R/M = 1).

Example The SBCD D0,D1 instruction subtracts the BCD quantity 11 contained in D1 from the BCD quantity 88 contained in register D0. The result is 77, replacing the contents of the destination register (specified by the Rx field). This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Register Rx			1	0	0	0	0	R/M	Register Ry		
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1

The hex value is $8101. Executing the instruction yields:

Before:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2704 -- -- SI PL=7 --Z--
D0: 00000088 00000011 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SBCD$       SBCD       D1,D0
```

After:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2700 -- -- SI PL=7 -----
D0: 00000077 00000011 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SBCD$+002   NOP
```

Scc Instruction

Set According to Condition

Syntax: Scc <ea>

Description

The Scc group of instructions sets a byte to either all zeros (false) or all ones (true) according to a combination of the CPU condition codes. These instructions allow storing the value of a comparison for use later (presumably when the condition codes have changed). A subsequent NEG.B operation on the destination operand of the Scc instruction will change the values to 0 for false and 1 for true, as opposed to $00 and $FF.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-		•				•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•
An	001	An		
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100		
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	Condition				1	1	Mode			Register		

Conditions The condition field (bits 11–8) indicates the combination of condition codes desired. Four bits allow sixteen possible combinations. The conditions and the corresponding instructions in the SCC family are as follows:

Condition	Meaning
0000 ($0)	**ST** (**S**et **T**rue). This instruction always sets the effective address.
0001 ($1)	**SF** (**S**et **F**alse). This instruction never sets the effective address.
0010 ($2)	**SHI** (**S**et **HI**gher). This instruction sets the effective address if both the Carry and Zero condition code bits are clear. **SHI** is equivalent to **S**et **G**reater **T**han (**SGT**) for unsigned numbers.
0011 ($3)	**SLS** (**S**et **L**ower or **S**ame). This instruction sets the effective address if either the Carry or Zero condition code bits are set. **SLS** is equivalent to **S**et **L**ess than or **E**qual (**SLE**) for unsigned numbers.
0100 ($4)	**SCC** (**S**et if **C**arry **C**lear). This instruction sets the effective address if the Carry bit is not set.
0101 ($5)	**SCS** (**S**et if **C**arry **S**et). This instruction sets the effective address if the Carry bit is set.
0110 ($6)	**SNE** (**S**et if **N**ot **E**qual). This instruction sets the effective address if the Z bit is not set.
0111 ($7)	**SEQ** (**S**et if **EQ**ual). This instruction sets the effective address if the Z bit is set.
1000 ($8)	**SVC** (**S**et if o**V**erflow **C**lear). This instruction sets the effective address if the V bit is not set.
1001 ($9)	**SVS** (**S**et if o**V**erflow **S**et). This instruction sets the effective address if the V bit is set.
1010 ($A)	**SPL** (**S**et if **PL**us). This instruction sets the effective address if the N bit is not set.
1011 ($B)	**SMI** (**S**et if **MI**nus). This instruction sets the effective address if the N bit is set.
1100 ($C)	**SGE** (**S**et if **G**reater than or **E**qual). This instruction sets the effective address if the N and V bits are both set, or if the N and V bits are both clear.
1101 ($D)	**SLT** (**S**et if **L**ess **T**han). This instruction sets the effective address if the N bit is set and the V bit is clear, or if the N bit is clear and the V bit is set.

1110 ($E)	**SGT** (**S**et if **G**reater **T**han). This instruction sets the effective address if the N and V bits are both set and the Z bit is clear, or if the N, V, and Z bits are all clear.
1111 ($F)	**SLE** (**S**et if **L**ess than or **E**qual). This instruction sets the effective address if the Z bit is set, if the N bit is set and the V bit is clear, or if the N bit is clear and the V bit is set.

Example The SNE D0 instruction sets the low-order byte of D0 to all ones if the Z condition code bit is clear. If the Z bit is set, the low-order byte of D0 is set to all zeros. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	Condition				1	1	Mode			Register		
0	1	0	1	0	1	1	0	1	1	0	0	0	0	0	0

The hex value of this instruction is $56C0. Execution yields:

Before:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SCC$       SNE       D0
```

After:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2700 -- -- SI PL=7 -----
D0: 000000ff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SCC$+002   NOP
```

STOP Instruction (Privileged)

Syntax: STOP #<data>

Load Status Register and Stop

Description

The STOP instruction places a 16-bit immediate value into the status register (SR) and suspends execution. An interrupt is required for the processor to continue. This instruction is normally used to suspend execution pending an interrupt. This instruction's ability to set all bits in the status register allows you to wait for an interrupt of an arbitrary priority. Specifying an interrupt mask of 7 causes the processor to wait for a non-maskable interrupt or an external RESET.

TRACE is the one exception to this process. It is possible to trace a STOP instruction without halting the processor. This technique is used in the example below.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*						•	•	•	•

X Loaded from bit 4 of immediate operand.
N Loaded from bit 3 of immediate operand.
Z Loaded from bit 2 of immediate operand.
V Loaded from bit 1 of immediate operand.
C Loaded from bit 0 of immediate operand.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0
T1	T2	S	M	–	I2	I1	I0	–	–	–	X	N	V	C	Z

The STOP instruction has hex value $4E72, followed by the immediate data value.

Example The STOP #$201F instruction clears the processor interrupt mask, sets all the condition code bits, and awaits an interrupt. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1

The hex value of this instruction is: $4E72 201F. Execution yields:

Before:

```
USP=b03f8692 MSP=b03f8692 ISP =b03f8690 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8690
PC =STOP$       STOP        #$201F
```

After:

```
USP=b03f8692 MSP=b03f8692 ISP =b03f8690 SR =201f -- -- SI PL=0 XNZVC
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8690
PC =STOP$+004   NOP
```

Note that an interrupt was necessary (in this case, a TRACE exception) for the processor to continue.

SUB Instruction

Subtract Binary

Syntax: SUB <ea>,Dn
SUB Dn,<ea>

Description

The SUB instruction performs a two's complement subtraction between a data register and an effective address operand.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•	•	•		•	•	•	•

X Set if a borrow was generated. Cleared otherwise.
N Set if the result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow was generated. Cleared otherwise.
C Set if a borrow was generated. Cleared otherwise.

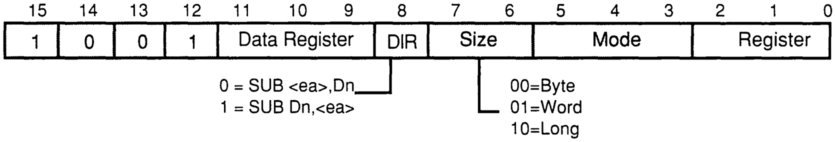
Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•		xxx.W	111	000	•	•
An *	001	An	•		xxx.L	111	001	•	•
(An)	010	An	•	•	#<data>	111	100	•	
(An)+	011	An	•	•					
-(An)	100	An	•	•					
(d16,An)	101	An	•	•	(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•	•	(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•	•	(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•	•	([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•	•	([bd,PC],Xn,od)	111	011	•	

* Word or long only ☐ 68020/030 only

When you specify a SUB instruction with an address register as a destination, many assemblers automatically generate the SUBA instruction. A similar situation exists with immediate data. You can specify an immediate mode source and a destination mode other than Dn, and the assembler will generate an SUBI instruction.

Instruction Format



Example The SUB.W D0,D1 instruction subtracts the low 16 bits of D0 from the low 16 bits of D1. The upper 16 bits of D1 are unchanged. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	Data Register			DIR	Size		Mode			Register		
1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0

The hex value of the instruction is $9240. Executing this instruction yields:

Before:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2700 -- -- SI PL=7 -----
D0: 00000001 03210000 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SUB$        SUB.W      D0,D1
```

After:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2719 -- -- SI PL=7 XN--C
D0: 00000001 0321ffff 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SUB$+002    NOP
```

Note the condition code settings. The X and C bits indicate a borrow from the seventeenth bit of the source, which does not exist for a 16-bit (word) operation.

SUBA Instruction

Subtract Address

Syntax: SUBA <ea>,An

Description

The SUBA instruction subtracts an effective address operand from an address register. Although the size may be specified as word or long, the entire address register is always affected. Both word and long operations produce a 32-bit result. A word operand is sign-extended to 32 bits and subtracted from the entire address register.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-			•	•		•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

The SUBA instruction does not change any condition code bits. You can perform address computations without losing the condition code settings of a previous data calculation.

Bits 11–9 contain the destination address register number. Bit

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•		xxx.W	111	000	•	
An	001	An	•	•	xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100	•	
(An)+	011	An	•						
-(An)	100	An	•						
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	Register An			Size	1	1	Mode			Register		

8 (Size) is zero for word operations and one for long operations.

Example The SUBA.W D0,A0 instruction sign-extends the low 16 bits of D0 and subtracts the result from all 32 bits of register A0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	Register An			Size	1	1	Mode			Register		
1	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0

The hex value of the instruction is $90C0. Executing the instruction yields:

Before:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2700 -- -- SI PL=7 -----
D0: 00000001 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 10000000 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SUBA$       SUBA.W     D0,A0
```

After:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2700 -- -- SI PL=7 -----
D0: 00000001 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 0fffffff 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SUBA$+002   NOP
```

SUBI Instruction

Syntax: SUBI #<data>,<ea>

Subtract Immediate

Description

The SUBI instruction subtracts immediate data from an effective address operand.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•	•	•		•	•	•	•

X Set if a borrow was generated. Cleared otherwise.
N Set if result is negative (bit 31/15/7= 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Set if an overflow was generated. Cleared otherwise.
C Set if a borrow was generated. Cleared otherwise.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•
An	001	An		
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100	•	
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	Size		Mode			Register		
Word Data (16 bits)								Byte Data (8 bits)							
Long Data															

The size field indicates the operation size: 00 for byte, 01 for word, and 10 for long. There are 2 or 4 bytes of immediate data that follow the instruction in memory. Byte and word operations have 16 bits (2 bytes) of immediate data. Long operations have 32 bits (4 bytes).

Example The instruction SUBI.L #$1000,D0 subtracts the hex quantity $1000 (4096 decimal) from all 32 bits of register D0. The immediate data ($1000) is contained in the two words following the instruction, as shown below:

Instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	Size		Mode			Register		
0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0

Data

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

This instruction has hex value $0480 0000 1000. Executing the instruction yields:

Before:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2700 -- -- SI PL=7 -----
D0: 00000fff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SUBI$      SUBI.L    #$00001000,D0
```

After:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2719 -- -- SI PL=7 XN--C
D0: ffffffff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SUBI$+006  NOP
```

SUBQ Instruction

Subtract Quick

Syntax: SUBQ #<data>,<ea>

Description

The SUBQ instruction subtracts a small immediate operand from an effective address. The immediate data can have the value 1–8. This instruction is 2 or 4 bytes shorter (depending on data size) than the equivalent SUBI instruction. SUBQ takes advantage of the fact that subtractions by a small constant are quite common, particularly in code generated by high-level languages. The presence of the SUBQ instruction makes this code smaller and faster than it otherwise might be.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•	•	•		•	•	•	•

X Set if a borrow was generated. Cleared otherwise.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Set if result is zero. Cleared otherwise.
V Set if an overflow was generated. Cleared otherwise.
C Set if a borrow was generated. Cleared otherwise.

When an address register is used as the destination, only word and long data sizes are allowed. In this situation, all 32 bits of the register are affected (regardless of instruction size). The condition codes are not affected.

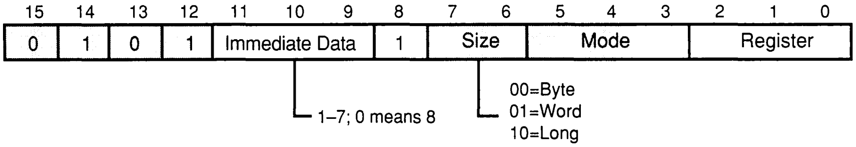
Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•
An *	001	An		•
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100	•	
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

* Word or long only ☐ 68020/030 only

Instruction Format


15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0
1
0
1
Immediate Data
1
Size
Mode
Register
1–7; 0 means 8
00=Byte
01=Word
10=Long

The immediate data field (bits 11–9) contains the data to be subtracted from the destination. A value of zero in this field is interpreted as 8 (subtracting zero is not very useful). The Size field (bits 7–6) contains 00 for adding bytes, 01 for words, and 10 for longs.

Example The SUBQ.L #1,D0 instruction decrements the 32-bit quantity in register D0 by 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	Immediate Data			1	Size		Mode			Register		
0	1	0	1	0	0	1	1	1	0	0	0	0	0	0	0

The hex value of this instruction is $5380. Executing the instruction yields:

Before:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2700 -- -- SI PL=7 -----
D0: 00000000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SUBQ$       SUBQ.L     #$1,D0
```

After:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2719 -- -- SI PL=7 XN--C
D0: ffffffff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SUBQ$+002   NOP
```

SUBX Instruction

Subtract BInary with eXtend

Syntax: **SUBX Dy,Dx**
SUBX -(Ay),-(Ax)

Description

The SUBX (SUBtract eXtended) instruction permits the subtraction of quantities that require more than 1 byte, word, or longword. This instruction is similar to a normal subtraction, except the eXtend condition code bit serves as a carry from the previous operation. The instruction can have two forms:

1. Subtract a data register from another data register
2. Subtract two memory locations using the Address Register Predecrement addressing mode

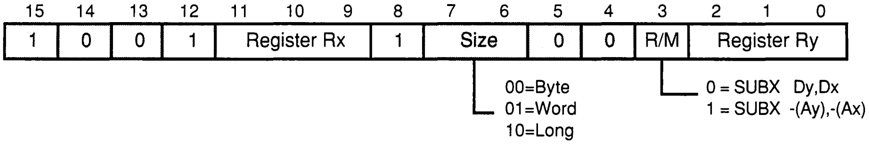
The 68000 big-endian data format dictates the form of multiple-precision operations with the SUBX instruction. The memory-to-memory form must begin at the highest-addressed byte and work back to the lowest-addressed byte, using predecrement addressing.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	*	*	*	*	*		•	•	•		•	•	•	•

X Set if a borrow was generated. Cleared otherwise.
N Set if result is negative (bit 31/15/7 = 1). Cleared otherwise.
Z Cleared if result was non-zero. Unchanged otherwise.
V Set if an overflow was generated. Cleared otherwise.
C Set if a borrow was generated. Cleared otherwise.

The behavior of the Z condition code follows the 68000 standard for multiple-precision arithmetic. You normally use a single-precision operation to subtract the low-order parts of multiple-precision operands. The Z bit will be set if this part of the operation is zero. If it is still set at the end of the operation, the entire result is zero.

Instruction Format


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1
0
0
1
Register Rx
1
Size
0
0
R/M
Register Ry
00=Byte
01=Word
10=Long
0 = SUBX Dy,Dx
1 = SUBX -(Ay),-(Ax)

Bits 11–9 contain the destination register number, and bits 2–0 contain the source register number. Bit 3 (R/M) is zero if the register numbers indicate data registers and one for address registers.

Example This example uses the SUBX instruction to subtract two pairs of data registers. The operands are each 64 bits (2 longwords) in length. The source operand is contained in the D3/D2 register pair, and the destination in the D1/D0 register pair. D2 and D0 contain the low-order portion of the operands.

This example requires two instructions: a SUB instruction to subtract the low-order 32 bits and a SUBX instruction to subtract the upper 32 bits. The layout of this latter instruction is:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td colspan="3">Immediate Data</td><td>1</td><td colspan="2">Size</td><td>0</td><td>0</td><td>R/M</td><td colspan="3">Register</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
</table>

The SUBX instruction has hex value $9383. Executing the instructions yields:

Before:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2700 -- -- SI PL=7 -----
D0: 0000000f 00000000 00000010 00000000 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SUBX$       SUB.L      D2,D0

USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2719 -- -- SI PL=7 XN--C
D0: ffffffff 00000000 00000010 00000000 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SUBX$+002   SUBX.L     D3,D1
```

After:

```
USP=b03f869a MSP=b03f869a ISP =b03f8698 SR =2719 -- -- SI PL=7 XN--C
D0: ffffffff ffffffff 00000010 00000000 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8698
PC =SUBX$+004   NOP
```

The SUB instruction subtracts the 2 low-order longwords, ignoring the state of the X condition code bit. Note that this instruction correctly sets the Z flag for all subsequent SUBX instructions. Setting the Z flag with a separate instruction is not required. The SUBX subtracts the high longwords, using the X bit generated by the SUB instruction. Note the carry between the low-order and high-order longwords.

SWAP Instruction

Swap Words of a Data Register

Syntax: SWAP Dx

Description

The SWAP instruction exchanges bits 31–16 with 15–0 of a data register.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0			•			•	•	•	•

X Not affected.
N Set if the result is negative (bit 31 is a one). Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	0	Register		

Example The SWAP D0 instruction exchanges the two halves of register D0. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	0	Data Register		
0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0

The hex value of the instruction is $4840. Execution yields:

Before:

```
USP=b03f862e MSP=b03f862e ISP =b03f862c SR =2700 -- -- SI PL=7 -----
D0: 12345678 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f862c
PC =SWAP$      SWAP      D0
```

After:

```
USP=b03f862e MSP=b03f862e ISP =b03f862c SR =2700 -- -- SI PL=7 -----
D0: 56781234 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f862c
PC =SWAP$+002   NOP
```

TAS Instruction

Syntax: TAS <ea>

Test And Set

Description

The TAS instruction provides a method for synchronizing multiple processors that share a memory area. This instruction tests a byte and sets the high-order bit to 1. The condition codes are set to the status of the byte before the operation. No other processor may access the byte until the entire operation is complete. This type of memory access is called a *read/modify/write* operation.

Multiprocessor synchronization requires each processor to issue a TAS instruction on a flag byte until the Z condition code bit becomes set, indicating that no other processor has the flag. Clearing the flag byte allows another processor to obtain exclusive access to the flag.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0		•				•	•	•	•

X Not affected.
N Set if bit 7 of the operand was set before the instruction began. Cleared otherwise.
Z Set if the operand was zero before the instruction began. Cleared otherwise.
V Always cleared.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		•
An	001	An		
(An)	010	An		•
(An)+	011	An		•
-(An)	100	An		•
(d16,An)	101	An		•
(d8,An,Xn)	110	An		•
(bd,An,Xn)	110	An		•
([bd,An,Xn],od)	110	An		•
([bd,An],Xn,od)	110	An		•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000		•
xxx.L	111	001		•
#<data>	111	100		
(d16,PC)	111	010		
(d8,PC,Xn)	111	011		
(bd,PC,Xn)	111	011		
([bd,PC,Xn],od)	111	011		
([bd,PC],Xn,od)	111	011		

☐ 68020/030 only

The TAS instruction has no source operand.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	Mode			Data Register		

Example The TAS D0 instruction performs a test and set operation on register D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	Mode			Register		
0	1	0	0	1	0	1	0	1	1	0	0	0	0	0	0

The instruction has hex value $4AC0. Execution yields:

Before:

```
USP=b03f862e MSP=b03f862e ISP =b03f862c SR =2700 -- -- SI PL=7 -----
D0: 00000080 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f862c
PC =TAS$        TAS        D0
```

After:

```
USP=b03f862e MSP=b03f862e ISP =b03f862c SR =2708 -- -- SI PL=7 -N---
D0: 00000080 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f862c
PC =TAS$+002    NOP
```

Note that the Z bit was not set, indicating that the operand was non-zero before the instruction.

TRAP Instruction

Take TRAP Exception

Syntax: TRAP #<vector>

Description

The TRAP instruction causes an exception through one of the TRAP vectors. There are 16 trap instructions, numbered 0–15. The corresponding vectors are vector numbers 32–47 (offsets $80–$BC). The TRAP exception uses a default format stack frame, shown in Figure 2-34. (Note that the 68000 default stack frame differs from the later chips.)

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-						•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	0	Vector			

Example The TRAP #15 instruction causes an exception through vector number 47 (offset $BC). This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	0	Vector			
0	1	0	0	1	1	1	0	0	1	0	0	1	1	1	1

The hex value of the instruction is $4E4F. Execution yields:

Before:

```
USP=b03f862e MSP=b03f862e ISP =b03f862c SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f862c
PC =TRAP$       TRAP        #$F
```

After:

```
USP=b03f862e MSP=b03f862e ISP =b03f8624 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8624
PC =TRAP15$     NOP
```

The stack frame looks like this:

```
b03f8624:       a700 b03f8f62 00bc
```

Note that the trace bit in the stacked status register is set, indicating that the TRAP instruction was traced. The return address ($B03F8F62) points to the instruction at TRAP$+2. On a 68000, the format word ($00BC) would not exist.

TRAPcc Instruction

TRAP on Condition

Syntax:

TRAPcc	
TRAPcc.W	**#<data>**
TRAPcc.L	**#<data>**

Description

The TRAPcc instruction causes an exception through vector 7 (offset $1C) if the specified condition is true. The TRAPcc exception uses a format 2 stack frame, shown in Figure 2-39. The Macintosh MPW system uses TPcc as the instruction mnemonic rather than TRAPcc. This instruction is not implemented on the 68000 or 68010.

TRAPcc instructions can be followed by one or two words of immediate data. The CPU adjusts the return address in the stack frame to skip over this immediate data upon executing an RTE. Note that the last two words in the stack frame (offsets $8 and $A) contain the address of the instruction word to assist in retrieving this immediate data.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-								•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	Condition				1	1	1	1	1	Opd	Size	

Opd:
0 => Operand
1 => No operand

Size:
10 => Word Operand
11 => Long Operand

Conditions The condition field (bits 11–8) indicates the combination of condition codes desired. Four bits allow 16 possible combinations. The conditions and the corresponding instructions in the TRAPcc family are

Condition	**Meaning**
0000 ($0)	**TRAPT** (**TRAP** **T**rue). This instruction always traps.
0001 ($1)	**TRAPF** (**TRAP** **F**alse). This instruction never traps.
0010 ($2)	**TRAPHI** (**TRAP** **HI**gher). This instruction traps if both the Carry and Zero condition code bits are clear. TRAPHI is equivalent to **TRAP** **G**reater **T**han (**TRAPGT**) for unsigned numbers.
0011 ($3)	**TRAPLS** (**TRAP** **L**ower or **S**ame). This instruction traps if either the Carry or Zero condition code bits are set. TRAPLS is equivalent to **TRAP** **L**ess **T**han or equal (**TRAPLE**) for unsigned numbers.
0100 ($4)	**TRAPCC** (**TRAP** if **C**arry **C**lear). This instruction traps if the Carry bit is not set.
0101 ($5)	**TRAPCS** (**TRAP** if **C**arry **S**et). This instruction traps if the Carry bit is set.
0110 ($6)	**TRAPNE** (**TRAP** if **N**ot **E**qual). This instruction traps if the Z bit is not set.
0111 ($7)	**TRAPEQ** (**TRAP** if **EQ**ual). This instruction traps if the Z bit is set.
1000 ($8)	**TRAPVC** (**TRAP** if o**V**erflow **C**lear). This instruction traps if the V bit is not set.
1001 ($9)	**TRAPVS** (**TRAP** if o**V**erflow **S**et). This instruction traps if the V bit is set.
1010 ($A)	**TRAPPL** (**TRAP** if **PL**us). This instruction traps if the N bit is not set.
1011 ($B)	**TRAPMI** (**TRAP** if **MI**nus). This instruction traps if the N bit is set.
1100 ($C)	**TRAPGE** (**TRAP** if **G**reater than or **E**qual). This instruction traps if the N and V bits are both set or if the N and V bits are both clear.
1101 ($D)	**TRAPLT** (**TRAP** if **L**ess **T**han). This instruction traps if the N bit is set and the V bit is clear or if the N bit is clear and the V bit is set.
1110 ($E)	**TRAPGT** (**TRAP** if **G**reater **T**han). This instruction traps if the N and V bits are both set and the Z bit is clear or if the N, V, and Z bits are all clear.
1111 ($F)	**TRAPLE** (**TRAP** if **L**ess than or **E**qual). This instruction traps if the Z bit is set, if the N bit is set and the V bit is clear or if the N bit is clear and the V bit is set.

Example The TRAPNE.L #$87654321 instruction causes an exception through vector number 7 (offset $1C) if the Z condition code bit is not set. This instruction has the following format:

Instruction

0	1	0	1	Condition				1	1	1	1	1	Opd	Size	
0	1	0	1	0	1	1	0	1	1	1	1	1	0	1	1

Data

1	0	0	0	0	1	1	1	0	1	1	0	0	1	0	1
0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1

The hex value of the instruction is $56FB 8765 4321. Execution yields:

Before:

```
USP=b03f862e MSP=b03f862e ISP =b03f862c SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f862c
PC =TRAPCC$     TRAPNE.L    #$87654321
```

After:

```
USP=b03f862e MSP=b03f862e ISP =b03f8620 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8620
PC =TRAP7$      NOP
```

Since the Z bit was not set, the exception took place. On entry to the exception handler (at label TRAP7$), the stack frame looks like this:

```
b03f8620: a700 b03f8f78 201c b03f8f72
```

Note that the trace bit in the stacked Status Register is set, indicating that the TRAPNE instruction was traced. The return address ($B03F8F78) points to the next instruction (at TRAPCC$+6). The instruction address ($B03F8F72) points to the TRAPNE instruction (at TRAPCC$).

TRAPV Instruction

Syntax: TRAPV

TRAP on Overflow

Description

The TRAPV instruction causes an exception through vector 7 (offset $1C) if the overflow bit is set. On the 68020 and later chips, the TRAPV exception uses a format 2 stack frame, shown in Figure 2-39. This instruction uses the default stack frame on the 68010 and 68000, as shown in Figure 2-34.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-						•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	0

Example The TRAPV instruction has hex value $4E76. Execution yields:

Before:

```
USP=b03f862e MSP=b03f862e ISP =b03f862c SR =2702 -- -- SI PL=7 ---V-
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f862c
PC =TRAPV$      TRAPV
```

After:

```
USP=b03f862e MSP=b03f862e ISP =b03f8620 SR =2702 -- -- SI PL=7 ---V-
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8620
PC =TRAP7$      NOP
```

Since the V bit was set, the exception took place. On entry to the exception handler (at label TRAP7$), the stack frame looks like this:

```
b03f8620:        a702 b03f8f82 201c b03f8f80
```

Note that the trace bit in the stacked status register is set, indicating that the TRAPV instruction was traced. The return address ($B03F8F82) points to the next instruction (at TRAPV$+4). The instruction address ($B03F8F80) points to the TRAPV instruction (at TRAPV$). This example was run on a 68030. The 68020 stack frame is the same. On a 68010, the instruction address part of the stack frame would not exist and the format code would be 0. On a 68000, only the status register and PC would be stacked (6 bytes).

TST Instruction

Syntax: TST <ea>

Test Operand

Description

The TST instruction sets the condition codes according to the value of an effective address operand. The effects of a TST instruction are the same as an immediate comparison (CMPI) to zero. TST is 2–4 bytes shorter, depending on the data size selected.

This instruction is required because the 68000 family has no Compare Quick instruction (similar to ADDQ or SUBQ). Even if there were such an instruction, the quick instruction format does not allow a value of zero.

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	*	*	0	0		•	•	•		•	•	•	•

X Not affected.
N Set if the operand is negative (Bit 31/15/7 = 1). Cleared otherwise.
Z Set if the operand is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•		xxx.W	111	000	•	
An	001	An			xxx.L	111	001	•	
(An)	010	An	•		#<data>	111	100		
(An)+	011	An	•						
-(An)	100	An	•					•	
(d16,An)	101	An	•		(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•		(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•		(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•		([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•		([bd,PC],Xn,od)	111	011	•	

☐ 68020/030 only

The TST instruction has no destination operand.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	Size		Mode			Register		

00=Byte
01=Word
10=Long

Example The TST D0 instruction sets the condition codes according to the contents of register D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	Size		Mode			Register		
0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0

The instruction has hex value $4A80. Execution yields:

Before:

```
USP=b03f862e MSP=b03f862e ISP =b03f862c SR =2700 -- -- SI PL=7 -----
D0: ffffffff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f862c
PC =TST$        TST.L      D0
```

After:

```
USP=b03f862e MSP=b03f862e ISP =b03f862c SR =2708 -- -- SI PL=7 -N---
D0: ffffffff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f862c
PC =TST$+002    NOP
```

UNLK Instruction

Unlink

Syntax: UNLK An

Description

The UNLK instruction reverses the action of a previous LINK instruction. Register A7 (the current stack pointer) receives the contents of An. The top longword on the stack is popped into An. This instruction is equivalent to the sequence:

```
MOVE.L    An,SP
MOVE.L    (SP)+,An
```

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-						•	•	•	•

X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	1	Register An		

Example The UNLK A6 instruction deallocates the stack frame whose base is in address register A6. The instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	1	Register An		
0	1	0	0	1	1	1	0	0	1	0	1	1	1	1	0

This instruction has hex value $4E5E. Execution yields:

Before:

```
USP=b03f862e MSP=b03f862e ISP =b03f861c SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 b03f8628 b03f861c
PC =UNLK$       UNLK       A6
```

A6 points to the new contents of A6:

```
b03f8628:       000000a6
```

After:

```
USP=b03f862e MSP=b03f862e ISP =b03f862c SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f862c
PC =UNLK$+002   NOP
```

UNPK Instruction

Unpack BCD

Syntax: UNPK -(Ax),-(Ay),#<data>
UNPK Dx,Dy,#<data>

Description

The UNPK instruction converts a single byte BCD quantity into two characters. This instruction is normally used to convert BCD into ASCII or EBCDIC digits. ASCII digits are in the range hex $30–$39 (0–9), and EBCDIC digits are in the range hex $F0–$F9 (0–9). The American Standard Code for Information Interchange (ASCII) is the dominant standard for characters currently. The older IBM EBCDIC (Extended Binary Coded Decimal Interchange Code) is widely used in IBM mainframe computers.

The UNPK instruction takes a byte quantity and transfers the upper and lower 4 bits (called nibbles) into the low 4 bits of the 2 bytes in a word. UNPK then adds an immediate value (2 bytes) to the word. The immediate value allows the use of a character set such as ASCII or EBCDIC. For ASCII, the immediate value should be $3030 (ASCII for 00). For EBCDIC, this value should be $F0F0 (EBCDIC for 00).

Condition Codes	X	N	Z	V	C	Sizes	Byte	Word	Long	CPUs	000	010	020	030
	-	-	-	-	-			•					•	•

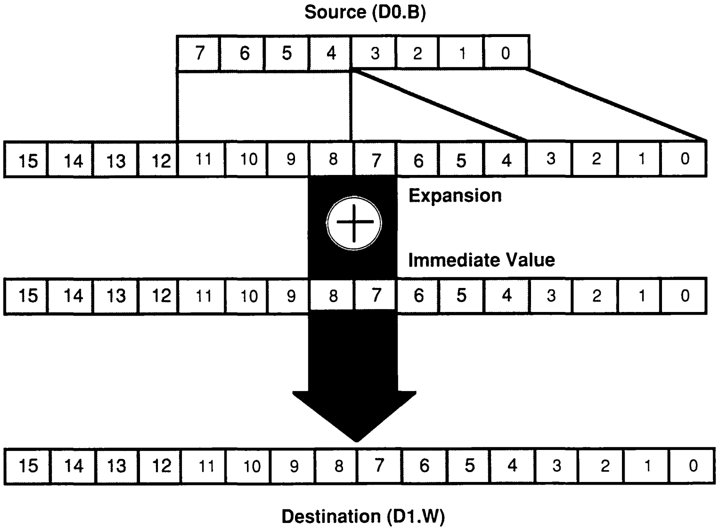
X Not affected.
N Not affected.
Z Not affected.
V Not affected.
C Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Register Ry			1	1	0	0	0	R/M	Register Rx		
16–bit immediate value															

Bit 3 (R/M) of the first instruction word is zero for the data register form of the instruction and one for the predecrement addressing mode form.

Example The UNPK D0,D1,#$3030 instruction converts the low byte of D0 to two ASCII digits, as shown below:


Source (D0.B)
7 6 5 4 3 2 1 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Expansion
Immediate Value
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Destination (D1.W)

This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Register Ry			1	1	0	0	0	R/M	Register Rx		
16-Bit Immediate Value															

1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	0

The instruction has hex value $8380 3030. Execution yields:

Before:

```
USP=b03f862e MSP=b03f862e ISP =b03f862c SR =2700 -- -- SI PL=7 -----
D0: 00000012 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f862c
PC =UNPK$       UNPK       D0,D1,#$3030
```

After:

```
USP=b03f862e MSP=b03f862e ISP =b03f862c SR =2700 -- -- SI PL=7 -----
D0: 00000012 00003132 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f862c
PC =UNPK$+004   NOP
```

$3132 is the ASCII value for the digits 12.

CHAPTER

4

Floating Point Instructions

"Numerical precision is the very soul of science."
— Sir D'Arcy Wentworth Thompson

Background

The 68020 and 68030 support an optional coprocessor chip that extends the basic 68000 instruction set to include floating point operations. There are two such chips, the 68881 and 68882. These chips differ primarily in speed, with the 68882 being slightly faster for certain instruction streams. Although applications software is completely compatible between the two chips, there are minor differences in systems software support for these two chips. Floating point data used by these two chips conforms to the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI-IEEE Std 754-1985), the most widely used standard in the microcomputer industry. The 68881/2 chips support a superset of this standard.

Architecture

On a machine with a floating point unit (FPU) installed, the programmer sees instructions that previously did not exist. The 68881 and 68882 conform to the coprocessor architecture described in Chapter 2.

Data Formats

The 68881/2 chips implement operations on 7 data types:

Name	Bits	Assembler ID
Byte Integer	8	.B
Word Integer	16	.W
Longword Integer	32	.L
Single Precision Real	32	.S
Double Precision Real	64	.D
Extended Precision Real	80(96)	.X
Packed Decimal String Real	96	.P

The extended precision real format occupies 96 bits in memory, but only 80 bits are actually used in operations. This situation is explained below.

All operations are performed in Extended format, with the other formats converted when transferred to or from memory. This technique makes mixed mode arithmetic quite simple.

Integer Data Formats The byte, word, and longword formats are identical to the integer formats used by the main CPU. These formats are described in Chapter 2. Using the integer data types for constants that have no fractional part saves program space because the integers are usually smaller than the floating point representation for the same quantity. Note, however, that the integer format is converted to an 80-bit extended floating point before being used in an instruction. Thus, there will be no difference in the numerical error of an operation regardless of whether integer or floating point data types were used to represent an integer constant.

Binary Floating Point Data Formats There are three binary representations for floating point data, as shown in Figure 4-1. A binary floating point number consists of a fraction (between 0.0 and 1.0) and a power of two by which the fraction is multiplied. The fraction is called the *mantissa* and the power of two is called the *exponent*. The sign of the number as a whole is always the uppermost bit. The exponent is an unsigned binary integer, with an implied bias. The bias values are 127, 1023, and 16383, for the Single, Double, and Extended formats, respectively. Exponents greater than the bias are considered positive, and exponents less than the bias are considered negative. This biased exponent is a

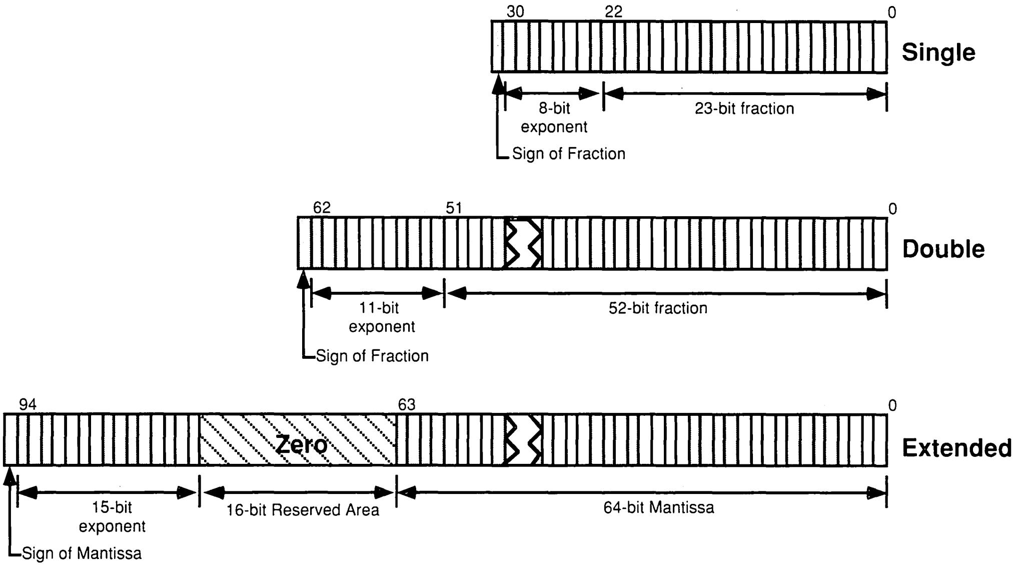

30
22
0
Single
8-bit exponent
23-bit fraction
Sign of Fraction
62
51
0
Double
11-bit exponent
52-bit fraction
Sign of Fraction
94
63
0
Zero
Extended
15-bit exponent
16-bit Reserved Area
64-bit Mantissa
Sign of Mantissa

Figure 4-1. 68881/2 Binary Floating Point Formats

power of two by which the fraction is multiplied to obtain the actual quantity desired. An interesting side effect of this representation is that you can use integer CMP instructions on floating point numbers in memory. (Note that only signed conditional operations will work, however.)

The data formats differ slightly. In order to retain the maximum amount of precision for a floating point quantity, the number is usually *normalized.* This means that the fractional part is shifted left and the exponent incremented until the most significant bit of the fraction is a one. The resulting number is in the range 1.0 to 2.0. Single and double precision representations, which must be normalized, store only the fractional part of the number. The integer part is assumed to be a single one bit, sometimes called the *hidden bit.* This technique yields an extra bit of precision. The extended format explicitly stores this bit (called the *integer bit*), forfeiting the extra bit of precision.

Binary floating point numbers can also be *denormalized.* When a binary format floating point number has an exponent of zero, the number is close to the minimum value permitted. In order to preserve the maximum possible range of values that can be represented, the upper bits in the mantissa are allowed to be zero. The single and double precision formats assume that the hidden bit is zero whenever the exponent is zero. In an extended format number, this bit is not hidden. If an extended format number has a one in the integer bit with zeros in the exponent, the number is still normalized. A zero in the integer bit means that the number is denormalized.

The extended format also permits a number with a zero integer bit and a non-zero exponent. Such a quantity is called an *unnormalized* number. The 68881 never generates such a result. Unnormalized numbers supplied as input to an operation are converted to normalized or denormalized form before use.

The extended format has 16 unused bits (79–64), which are required to be zero. These bits are reserved for future expansion and to allow floating point numbers to be longword-aligned. An extended format floating point number occupies three longwords in memory.

Packed Decimal String Real Data Format The packed decimal real format is shown in Figure 4-2. This representation has a 17-digit mantissa, a 3-digit exponent, and 2 sign bits. The digits are all packed BCD, compatible with the format used by the main processor. In this format, both the exponent and mantissa have sign bits.

The top 4 bits (95–92) consist of the 2 sign bits and 2 bits that are used only for Infinity or Not A Number (NAN) values (described below). The exponent occupies the next 3 digits (12 bits). The next 12 bits are unused, except when converting a number with an exponent larger than 999. In this case, bits 79–76 are used as the fourth digit of the exponent. These bits are unused at all other times. The remainder of the number contains 17 digits for the mantissa.

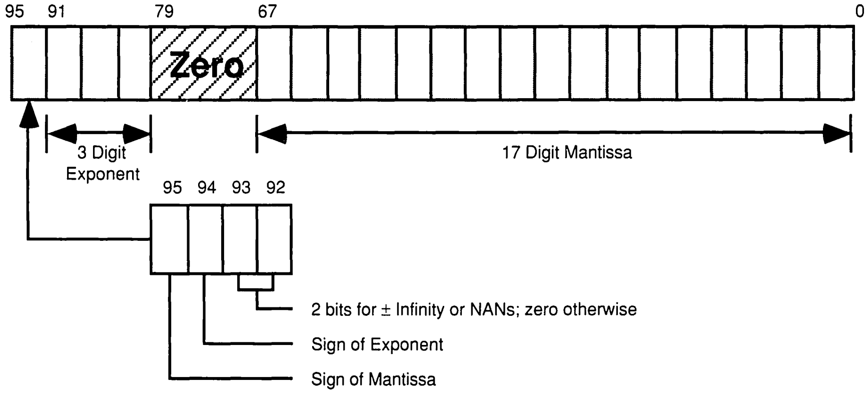


Figure 4-2. Packed Decimal Real Format

Note that integer comparison operations do not work with this format as with the binary real formats.

Internal Data Format All 68881/2 operations are performed in extended precision format. Any operands not in this format are converted prior to the operation. All internal registers (described below) use the extended precision format shown in Figure 4-1. This format has a 15-bit biased exponent and 64 bits of mantissa.

Intermediate quantities computed during an operation have a slightly enhanced format to make overflow detection and rounding simpler. This internal format is shown in Figure 4-3.

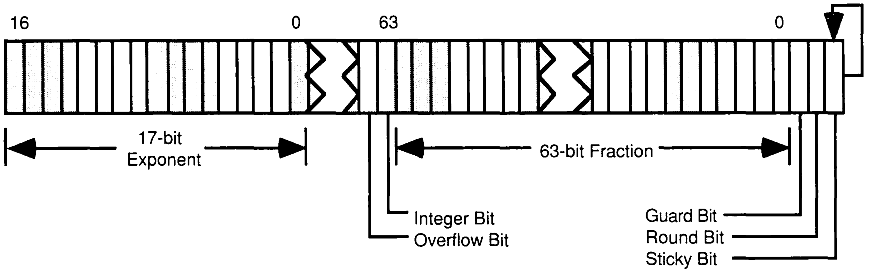


Figure 4-3. Internal Data Format

There are two additional bits in the exponent. One is for a bit of precision and one for a sign bit. The exponent in the internal format is two's complement rather than biased. The mantissa has three additional bits: a *guard bit*, which adds an extra bit of precision, a *round bit*, which permits easier rounding, and a *sticky bit*, which preserves the non-zero aspect of very small numbers.

The 68881/2 coprocessors support four rounding modes, in compliance with the IEEE standard. These modes are as follows:

- **Round to Nearest.** The value closest to an infinite precision result is chosen. In case of a tie, the even number is chosen. This technique is sometimes called "*round to nearest, even.*"
- **Round to Zero.** The value closest to, but whose absolute value is not greater than, the infinitely precise result is chosen. This technique is sometimes called "*chop mode*," since the implementation simply zeros all bits to the right of the rounding point.
- **Round to Minus Infinity.** The value closest to, but not greater than, the infinitely precise result (possibly minus infinity) is chosen.

- **Round to Positive Infinity.** The value closest to, but not less than, the infinitely precise result (possibly positive infinity) is chosen.

The round to infinity modes are sometimes called "*directed rounding modes*," and are useful for interval arithmetic.

Rounding takes place using the intermediate data format shown in Figure 4-3. If a format other than extended is selected for the operation, the guard, round, and sticky bits move to the left to allow reduced precision operation. The guard and round bits are always calculated exactly. The sticky bit is the logical OR of all bits to the right of the round bit in the theoretical infinite-precision result. This allows extra precision in the rounding process. The result of a single computation in round to nearest mode is in error by no more than one-half unit in the least-significant place. Note that this precision is for operations, not instructions. Transcendental functions are composed of a set of operations and can have errors up to one unit of the low-order bit of double precision (4096 units in the last bit of extended precision). Typical error for these instructions is 64 units in the last bit of extended precision, however.

Infinity Formats The IEEE specification incorporates a data format for numbers larger than the maximum representable value. This condition is termed an *overflow* and occurs whenever the exponent value of a result is too large to fit the data format. For binary floating point in single or double precision format, the exponent is set to the maximum, and the fraction must be zero. For extended precision, the exponent is set to the maximum, and the low order 63 bits of the mantissa are set to zero. The most significant bit can be either zero or one. The sign bit of all three forms can be 0 or 1, allowing representation of both positive or negative infinity. This representation appears in Figure 4-4.

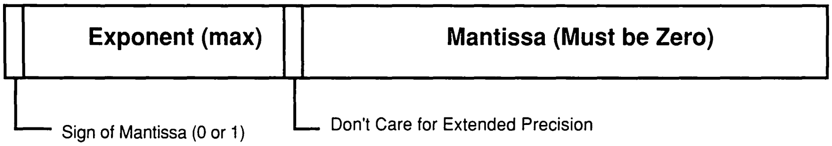


Figure 4-4. Binary Representation of Infinity

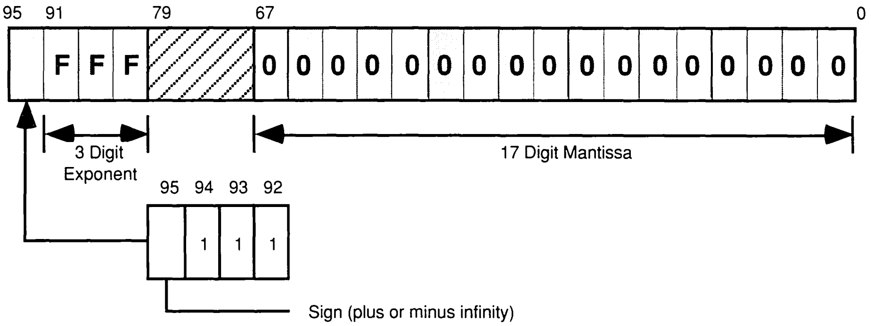


Figure 4-5. BCD Representation of Infinity

The representation of infinity in the BCD format appears in Figure 4-5.

The exponent (bits 91–80) must equal $FFF, and the mantissa must be all zeros. Bits 94–92 must all be one, and bit 95 gives the sign of the infinite quantity. Bits 68–79 (zero in normal BCD numbers) are Don't Care and can have any value.

Not A Number (NAN) Formats The IEEE specification provides a way to describe the result of an operation that is not meaningful. Such a quantity is called Not A Number (NAN) in IEEE parlance. A NAN is the result of dividing infinity by infinity, for instance. All operations where one of the operands is a NAN yield a result of NAN.

A second form of NAN is the *signaling* NAN, or SNAN. When the 68881/2 attempts to use a signaling NAN in an operation, an exception through vector 54 results. You can use a signaling NAN to implement data types (in software) other than the standard IEEE format data types. Signaling NANs are never generated by the Motorola coprocessors.

A binary format NAN is similar to a binary format infinity, except the mantissa must be non-zero rather than zero for infinity. As in the representation for infinity, bit 63 of the mantissa of an extended precision format number can be either zero or one. Thus representing a NAN in extended precision requires that at least one of bits 62–0 be non-zero. This format is shown in Figure 4-6.

The exponent must be the maximum value for the format (single, double, or extended). The most significant bit of the mantissa (except for extended format) determines whether a NAN is a signaling NAN or not. The second most significant bit makes this determination for extended format. This bit is zero for a signaling NAN and one for a non-signaling NAN.

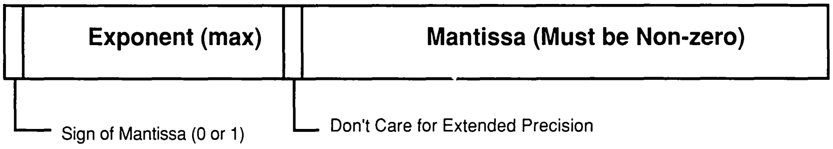

Exponent (max)
Mantissa (Must be Non-zero)
Sign of Mantissa (0 or 1)
Don't Care for Extended Precision

Figure 4-6. Binary Representation NAN

A BCD format NAN quantity is similar to the BCD format for infinity, except the mantissa must be non-zero. Mantissa bit 66 indicates whether the NAN is signaling or non-signaling, as shown in Figure 4-7.

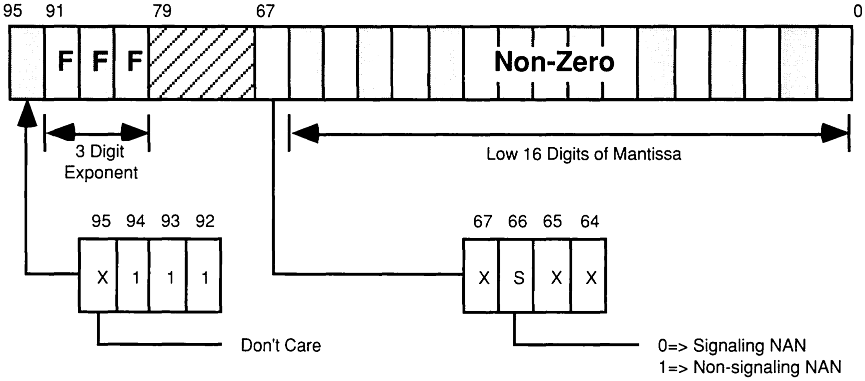

95 91 79 67 0
F F F
Non-Zero
3 Digit Exponent
Low 16 Digits of Mantissa
95 94 93 92
X 1 1 1
Don't Care
67 66 65 64
X S X X
0=> Signaling NAN
1=> Non-signaling NAN

Figure 4-7. BCD NAN Format

Register Set

The 68881 and 68882 coprocessors have eleven programmer-visible registers. There are eight floating point data registers, a control register, a status register, and an instruction address register. These registers appear in Figure 4-8.

The eight floating point data registers are extended floating point numbers without the 16 wasted bits to the left of the mantissa. All operands are converted to extended precision before being used by the coprocessor.

Floating Point Control Register (FPCR) The 16-bit floating point control register contains two separate 8-bit areas. One area enables and disables exceptions, and one area controls rounding in floating point operations. These bytes are called the *FPCR Exception Enable Byte* and the *FPCR Mode Control Byte*, respectively.

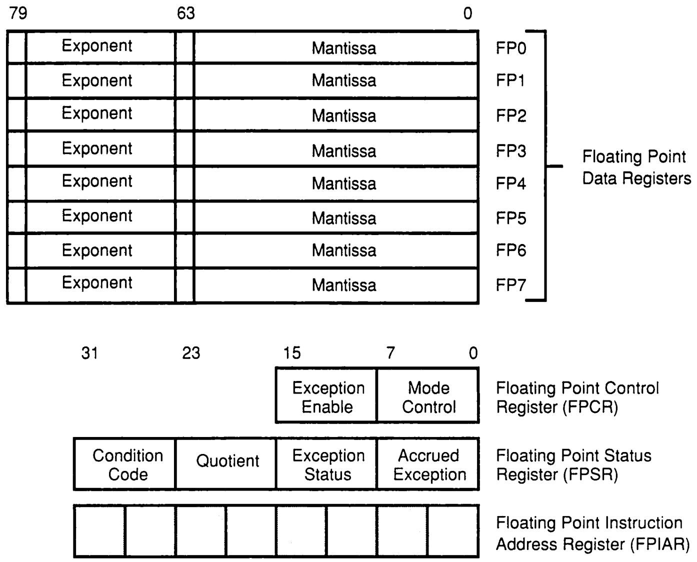

79
63
0
Exponent
Mantissa
FP0
FP1
FP2
FP3
FP4
FP5
FP6
FP7
Floating Point Data Registers
31
23
15
7
0
Exception Enable
Mode Control
Floating Point Control Register (FPCR)
Condition Code
Quotient
Exception Status
Accrued Exception
Floating Point Status Register (FPSR)
Floating Point Instruction Address Register (FPIAR)

Figure 4-8. 68881/68882 Registers

The Exception Enable Byte has a bit that controls each of the maskable exceptions in the coprocessor. This byte appears in Figure 4-9.

Bit 15 of the FPCR is the Branch/Set on Unordered (BSUN) enable bit. This condition is the result of a conditional operation (FBcc, FDBcc, FScc or FTRAPcc instruction) where one of the inputs to the preceding arithmetic operation was a NAN. Setting this bit means that the coprocessor will cause an exception through vector 48 on this condition.

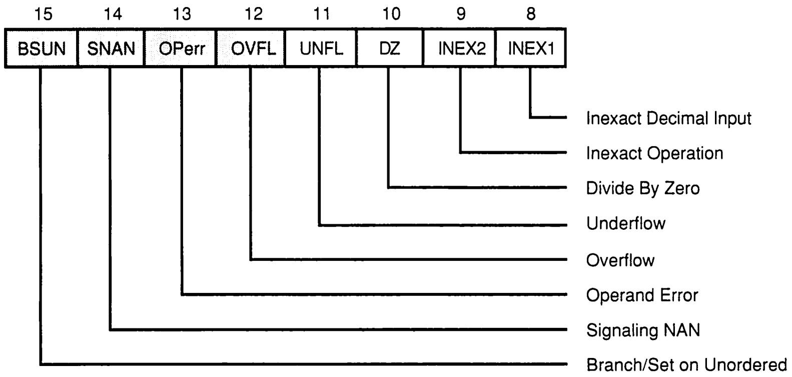

15
14
13
12
11
10
9
8
BSUN
SNAN
OPerr
OVFL
UNFL
DZ
INEX2
INEX1
Inexact Decimal Input
Inexact Operation
Divide By Zero
Underflow
Overflow
Operand Error
Signaling NAN
Branch/Set on Unordered

Figure 4-9. FPCR Exception Enable Byte

Bit 14 of the FPCR is the Signaling NAN (SNAN) enable bit. Setting this bit means that the coprocessor will cause an exception when fetching an SNAN from memory. This signaling NAN exception uses vector 54. You can implement non-standard data types in software, using a signaling NAN with the data contained in the mantissa bits. The mantissa must be non-zero, however.

Bit 13 of the FPCR is the operand error (OPERR) enable bit. Setting this bit causes the coprocessor to generate an exception through vector 52 on an operand error. Such an error results when data for an instruction is meaningless (for example, adding positive infinity to negative infinity).

Bit 12 of the FPCR is the Overflow (OVFL) enable bit. Setting this bit causes the coprocessor to generate an exception through vector 53 when the result of an operation is too large to represent in a floating point register or in a selected memory format.

Bit 11 of the FPCR is the Underflow (UNFL) enable bit. Setting this bit causes the coprocessor to generate an exception through vector 51 when the result of an operation cannot be represented either in a register or in memory as a normalized floating point number.

Bit 10 of the FPCR is the Divide by Zero (DZ) enable bit. Setting this bit causes the coprocessor to generate an exception through vector 50 whenever a floating point division by zero is attempted. Note that some of the transcendental functions can cause this error as well. (See the section on Exceptions for details.)

Bit 9 of the FPCR is the Inexact Result (INEX2) enable bit. Setting this bit causes the coprocessor to generate an exception through vector 49 if an operation generates any intermediate result that cannot be represented exactly in the selected rounding precision or in the destination data format.

Bit 8 of the FPCR is the Inexact Decimal Conversion (INEX1) enable bit. Setting this bit causes the coprocessor to generate an exception through vector 49 if conversion from decimal input yields an inexact result.

The mode control byte of the FPCR appears in Figure 4-10. The values selected for the register fields are such that zero results in selection of the IEEE default modes.

This byte controls rounding of floating point results for all but a few instructions that force a particular rounding configuration. Bits 7–6 regulate the precision of the final result of an operation. You can select extended (64-bit mantissa), single (32-bit mantissa), or double (53-bit mantissa). This precision affects only the final result of an operation. All intermediate results use extended precision.

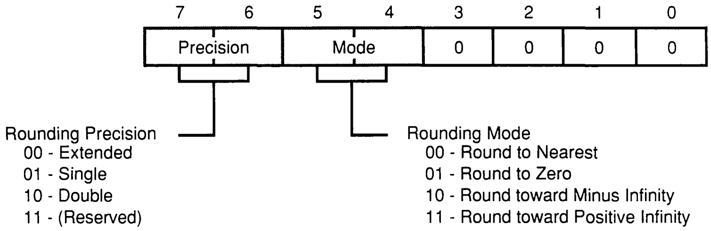

7
6
5
4
3
2
1
0
Precision
Mode
0
0
0
0
Rounding Precision
00 - Extended
01 - Single
10 - Double
11 - (Reserved)
Rounding Mode
00 - Round to Nearest
01 - Round to Zero
10 - Round toward Minus Infinity
11 - Round toward Positive Infinity

Figure 4-10.4 FPCR Mode Control Byte

Bits 5–4 control the method used in rounding inexact results. Round to nearest mode means that the rounded value is the number closest to the infinitely precise result. In case of a tie, the even value is selected. Round to zero truncates the result to the selected size (in bits 7–6). Round toward plus and minus infinity modes select the value closer to plus or minus infinity, respectively. A more detailed description of rounding techniques appears in the Intermediate Data Format section above.

Floating Point Status Register (FPSR) The floating point status register contains four bytes that are modified by most floating point instructions. These bytes provide an indication of the floating point unit status at any given point in time. These 4 bytes are termed the Condition Code byte, the Quotient byte, the Exception Status byte, and the Accrued Exception byte. All bits of this register can be read or written explicitly with an FMOVE instruction.

The FPSR condition code byte appears in Figure 4-11.

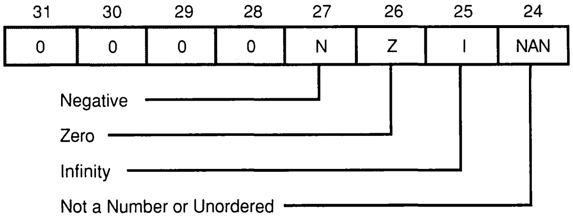

31
30
29
28
27
26
25
24
0
0
0
0
N
Z
I
NAN
Negative
Zero
Infinity
Not a Number or Unordered

Figure 4-11. Floating Point Status Register Condition Code Byte

It is possible to determine the setting of the condition codes by inspection of the result itself. The codes are set as follows:

Result Type	N	Z	I	NAN
+ Normalized or Denormalized	0	0	0	0
- Normalized or Denormalized	1	0	0	0
+ 0	0	1	0	0
- 0	1	1	0	0
+ Infinity	0	0	1	0
- Infinity	1	0	1	0
+ NAN	0	0	0	1
- NAN	1	0	0	1

Note that of the sixteen possible combinations of these four bits, only the eight combinations above can be generated by 68881/2 instructions. You can set these bits to any value using the FMOVE instruction. Behavior of conditional instructions on condition combinations other than those listed above is not guaranteed. The interpretation of these codes is described below in the section on Floating Point Conditions.

The FPSR quotient byte is used on the floating point modulo (FMOD) and IEEE remainder (FREM) instructions. This byte is shown in Figure 4-12.

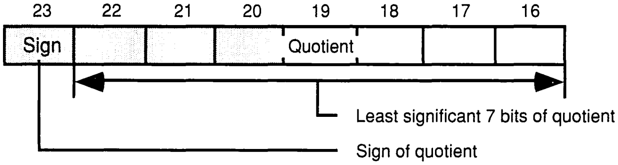


Figure 4-12. Floating Point Status Register Quotient Byte

Bit 23 contains the sign of the quotient, and bits 22–16 contain the low order seven bits of the quotient. One use of this information is for argument reduction in transcendental functions. The quadrant of a circle in which an operand resides can readily be obtained from this information. The quotient byte is set by the FMOD and FREM instructions and is unchanged unless explicitly accessed by the user program.

The FPSR exception status byte indicates which floating point exceptions are pending. This byte appears in Figure 4–13.

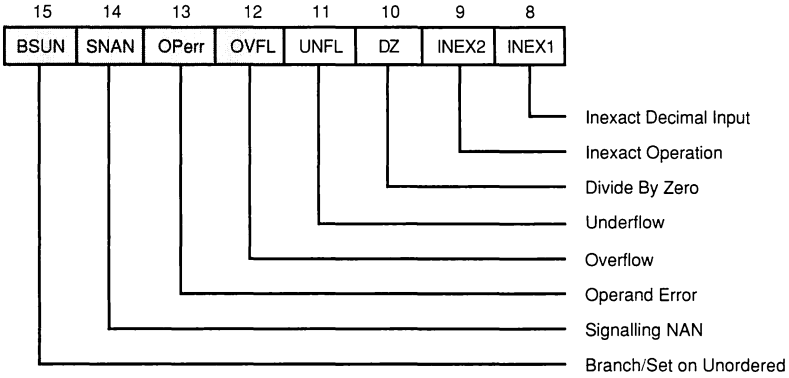

15
14
13
12
11
10
9
8
BSUN
SNAN
OPerr
OVFL
UNFL
DZ
INEX2
INEX1
Inexact Decimal Input
Inexact Operation
Divide By Zero
Underflow
Overflow
Operand Error
Signalling NAN
Branch/Set on Unordered

Figure 4-13. Floating Point Status Register Exception Status Byte

These bits correspond to the exception enable bits described in Figure 4-9. Note that even the bit positions in the register are the same. The difference is that the exception status byte is set whenever the conditions that might cause an exception occur. The exception enable byte allows you to select which conditions can cause a main CPU exception. Thus, the exception occurs whenever a bit in the exception status byte in the FPSR becomes set as the result of a computation and the corresponding bit is set in the exception enable byte of the FPCR. Setting these bits with an FMOVE instruction does not cause an exception to occur.

The Floating Point Status Register Accrued Exception byte allows testing for an exception condition at the end of a series of floating point instructions. This feature eliminates the need for taking an exception on an unusual condition and for checking the exception status byte following each instruction. Any exception conditions are registered as they occur, and none of the flags are cleared, except by the user program. This allows a single check at the end of a series of computations.

The Accrued Exception byte is shown in Figure 4-14.

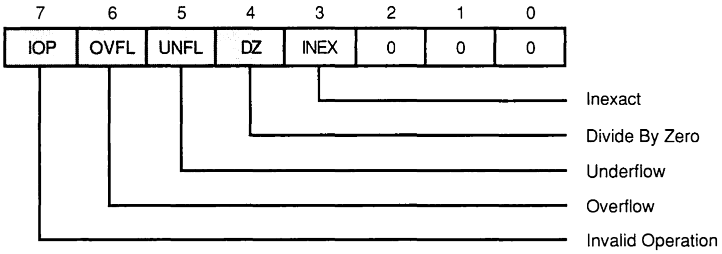

7
6
5
4
3
2
1
0
IOP
OVFL
UNFL
DZ
INEX
0
0
0
Inexact
Divide By Zero
Underflow
Overflow
Invalid Operation

Figure 4-14. Floating Point Status Register Accrued Exception Byte

The IOP bit is set anytime the BSUN, SNAN, or OPERR bits are set by an instruction. The OVFL bit is set if the Exception Status OVFL bit ever becomes set. The UNFL bit is set if both the UNFL and INEX2 bits are set simultaneously in the Exception Status byte. The DZ bit latches the DZ bit in the Exception Status byte. The INEX bit is set if any of the Exception Status INEX1, INEX2, or OVFL bits become set. These bits remain set until an FMOVE instruction clears them, or the coprocessor is reset.

Floating Point Instruction Address Register (FPIAR) The Floating Point Instruction Address Register provides assistance in locating the instruction that caused an exception. Since the main CPU can process instructions concurrently with the coprocessor, the PC stacked with an exception stack frame may not point to the instruction that caused the exception. The 68882 can process two floating point instructions concurrently, which further obscures the culprit instruction. When the coprocessor has any exceptions enabled in the floating point control register, the FPIAR is loaded with the address of each instruction before it begins execution. Instructions that cannot cause an exception do not disturb the FPIAR.

Floating Point Conditional Operations

The 68881/2 coprocessor chips support a series of instructions that depend on a condition that is evaluated by the coprocessor. These instructions are FBcc, FDBcc, FScc, and FTRAPcc. They are analogous to the Bcc, DBcc, Scc, and TRAPcc integer instructions, except the condition is based on the floating point condition codes shown in Figure 4-11.

Each conditional instruction has a 6-bit field that specifies the combination of condition code bits to be evaluated. This code resides in bits 5–0 of the second word of the instruction, except for the FBcc instruction, in which the code resides in bits 5–0 of the first instruction word. Bit 5 of this field must be zero, giving 32 possibilities, as shown in Figure 4-15.

The formulas for the condition codes use the following C language notation: "&" is Logical And, "|" is Logical OR, and "~" is Logical Not. Parentheses indicate groupings just as in C. Values 0 and 1 mean "always false" and "always true," respectively.

The presence of quantities that are "Not a Number" complicates the condition codes. If comparing a valid number to a NAN is

Mnemonic	Mnemonic Expansion	Equation	Predicate
F	False	0	000000
EQ	Equal	Z	000001
OGT	Ordered Greater Than	~(NAN | Z | N)	000010
OGE	Ordered Greater Than or Equal	Z | ~(NAN | N)	000011
OLT	Ordered Less Than	N & ~(NAN | Z)	000100
OLE	Unordered or Less Than or Equal	Z | (N & ~NAN)	000101
OGL	Ordered Greater or Less Than	~(NAN | Z)	000110
OR	Ordered	~NAN	000111
UN	Unordered	NAN	001000
UEQ	Unordered or Equal	NAN | Z	001001
UGT	Unordered or Greater Than	NAN | ~(N | Z)	001010
UGE	Unordered or Greater Than or Equal	NAN | Z | N	001011
ULT	Unordered or Less Than	NAN | (N & ~Z)	001100
ULE	Unordered or Less Than or Equal	NAN | Z | N	001101
NE	Not Equal	~Z	001110
T	True	1	001111
SF	Signaling False	0	010000
SEQ	Signaling Equal	Z	010001
GT	Greater Than	~(NAN | Z | N)	010010
GE	Greater Than or Equal	Z | ~(NAN | N)	010011
LT	Less Than	N & ~(NAN | Z)	010100
LE	Less Than or Equal	Z | (N& ~NAN)	010101
GL	Greater or Less Than	~(NAN | Z)	010110
GLE	Greater, Less, or Equal	~NAN	010111
NGLE	Not (Greater, Less, or Equal)	NAN	011000
NGL	Not (Greater or Less Than)	NAN | Z	011001
NLE	Not (Less Than or Equal)	NAN | ~(N | Z)	011010
NLT	Not Less Than	NAN | (Z | ~N)	011011
NGE	Not (Greater Than or Equal)	NAN | (N & ~Z)	011100
NGT	Not Greater Than	NAN | Z | N	011101
SNE	Signaling Not Equal	~Z	011110
ST	Signaling True	1	011111

▭ Can set Status Register BSUN exception bit

Figure 4-15. Floating Point Conditions

allowed, the opposite of "Greater Than" is not "Less Than or Equal." It is "Not Greater Than." The conditions are also divided into two groups: those that can cause a BSUN exception and those which cannot. The non-exception group are known as the "IEEE aware," and the exception group as the "IEEE Non-Aware." These exceptions are grouped as follows:

IEEE Aware		IEEE Non-Aware	
F	000000	SF	010000
T	001111	ST	011111
EQ	000001	SEQ	010001
NE	001110	SNE	011110
OGT	000010	GT	010010
ULE	001101	NGT	011101
OGE	000011	GE	010011
ULT	001100	NGE	011100
OLT	000100	LT	010100
UGE	001011	NLT	011011
OLE	000101	LE	010101
UGT	001010	NLE	011010
OGL	000110	GL	010110
UEQ	001001	NGL	011001
OR	000111	GLE	010111
UN	001000	NGLE	011000

Note that bit 4 determines whether a condition can cause an exception. The Aware and Non-Aware tests shown above differ only in whether or not the BSUN exception is raised if the NAN bit in the Floating Point Condition Code byte is set.

The **F** (**F**alse) and **SF** (**S**ignaling **F**alse) conditions are always false. An operation that uses this condition combination is a No-Operation (**NOP**). Note that there is a **F**loating Point **NOP** (**FNOP**) instruction. The Signaling False condition will set the BSUN bit if the NAN condition code bit is set.

The **T** (**T**rue) and **ST** (**S**ignaling **T**rue) conditions are always true. Any conditional instruction that must always execute uses this condition combination. The Signaling True condition will set the BSUN bit if the NAN bit is set.

The **EQ** (**Eq**ual) and **SEQ** (**S**ignaling **Eq**ual) conditions are true if the Z bit in the FPSR is set. The Signaling Equal condition will set the BSUN bit if the NAN bit is set.

The **NE** (**N**ot **E**qual) and **SNE** (**S**ignaling **N**ot **E**qual) conditions are true if the Z bit in the FPSR is clear. The Signaling Equal condition will set the BSUN bit if the NAN bit is set.

The **OGT** (**O**rdered **G**reater **T**han) condition is true if the NAN, Z, and N condition codes are all zero. This condition cannot raise the BSUN bit. The **GT** (**G**reater **T**han) condition is the same comparison without the presence of the NAN bit. This condition is true if both the Z and N bits are zero. If the NAN bit is set, the GT condition code will set the BSUN bit, possibly causing an exception.

The **ULE** (**U**nordered or **L**ess than or **E**qual) condition is the opposite of the OGT condition. This condition is true if any of the NAN, Z, or N condition codes is a one. This condition cannot raise the BSUN bit. The **LE** (**L**ess than or **E**qual) condition is the same comparison without the presence of the NAN bit. This condition is true if either the Z or N bits is one. If the NAN bit is set, the LE condition will set the BSUN bit, possibly causing an exception.

The **OGE** (**O**rdered **G**reater than or **E**qual) condition is true if the Z bit is set or if both the NAN and N bits are zero. This condition cannot raise the BSUN bit. The **GE** (**G**reater than or **E**qual) condition is the same comparison without the presence of the NAN bit. This condition is true if either the Z bit is set or the N bits is clear. If the NAN bit is set, the GE condition will set the BSUN bit, possibly causing an exception.

The **ULT** (**U**nordered or **L**ess **T**han) and **NGE** (**N**ot **G**reater than or **E**qual) conditions are true if one of the NAN, Z, or N bits are set. If the NAN bit is set, the NGE condition will set the BSUN bit, possibly causing an exception.

The **OLT** (**O**rdered **L**ess **T**han) and **LT** (**L**ess **T**han) conditions are true if the N bit is set and the NAN and Z bits are both clear. If the NAN bit is set, the LT condition will set the BSUN bit, possibly causing an exception.

The **UGE** (**U**nordered or **G**reater than or **E**qual) condition is true if any of the NAN, Z, or N bits is set. The **NLT** (Not Less Than) condition is true if the NAN bit is set or the Z bit is set and the N bit is clear. If the NAN bit is set, the NLT condition will set the BSUN bit, possibly causing an exception.

The **OLE** (**O**rdered **L**ess than or **E**qual) and **LE** (Less Than or Equal) conditions are true if the Z bit is set, the N bit is set and the NAN bit is clear. If the NAN bit is set the LE condition will set the BSUN bit, possibly causing an exception.

The **UGT** (**U**nordered or **G**reater **T**han) and **NLE** (**N**ot **L**ess than or **E**qual) conditions are true if the NAN bit is set or both the Z, and N bits are clear. If the NAN bit is set, the NLE condition will set the BSUN bit, possibly causing an exception.

The **OGL** (**O**rdered **G**reater than or **L**ess than) and **GL** (**G**reater than or **L**ess than) conditions are true if the Z and NAN bits are both clear. If the NAN bit is set, the GL condition will set the BSUN bit, possibly causing an exception.

The **UEQ** (**U**nordered or **Eq**ual) and **NGL** (**N**ot **G**reater than or **L**ess than) conditions are true if either the NAN or Z bit is set. If the NAN bit is set, the NGL condition will set the BSUN bit, possibly causing an exception.

The **OR** (**Or**dered) and **GLE** (**G**reater than or **L**ess than or **E**qual) conditions are true if the NAN bit is clear. If the NAN bit is set, the GLE condition will set the BSUN bit, possibly causing an exception.

The **UN** (**Un**ordered) and **NGLE** (**N**ot **G**reater than or **L**ess than or **E**qual) conditions are true if the NAN bit is set. If the NAN bit is set, the NGLE condition will set the BSUN bit, possibly causing an exception.

Floating Point State Frames

The FSAVE and FRESTORE instructions save and restore the internal operations of the coprocessor. The internal state is saved in a memory area called a *state frame*. There are three separate formats for this information. These formats differ between the 68881 and 68882 chips.

Figure 4-16 contains the state frames for the 68881 chip.

Figure 4-17 shows the 68882 state frame formats.

In all state frames on both coprocessors, reserved or undefined bits are written as ones (during an FSAVE instruction) and ignored on reads (during an FRESTORE instruction). The first byte of the state frame is a coprocessor version number, which for IDLE and BUSY frames can be $1F or $3F for the 68881 and $1F for the 68882. For the NULL frame format, this byte must be zero. For the NULL and BUSY state frames, the second byte gives the length of the frame, not including the first longword. Note that the 68881 and 68882 formats of these frames have different lengths and different organizations.

Null State Frame In both chips, the null state frame is 4 bytes long. The first byte must be zero, and the other three are ignored. This state frame is extremely useful in initializing the coprocessor to a known benign condition. A state frame with a version byte of zero is guaranteed to work on all future versions of the 6888x

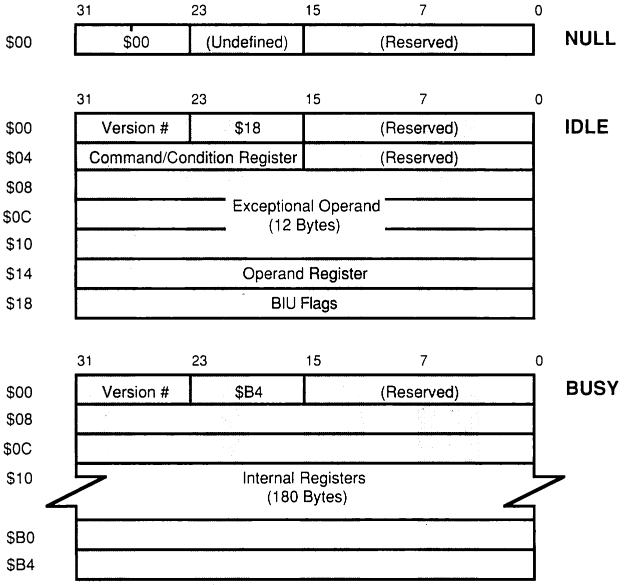

31
23
15
7
0
$00
$00
(Undefined)
(Reserved)
NULL
31
23
15
7
0
$00
Version #
$18
(Reserved)
IDLE
$04
Command/Condition Register
(Reserved)
$08
$0C
Exceptional Operand
(12 Bytes)
$10
$14
Operand Register
$18
BIU Flags
31
23
15
7
0
$00
Version #
$B4
(Reserved)
BUSY
$08
$0C
$10
Internal Registers
(180 Bytes)
$B0
$B4

Figure 4-16. 68881 State Frames

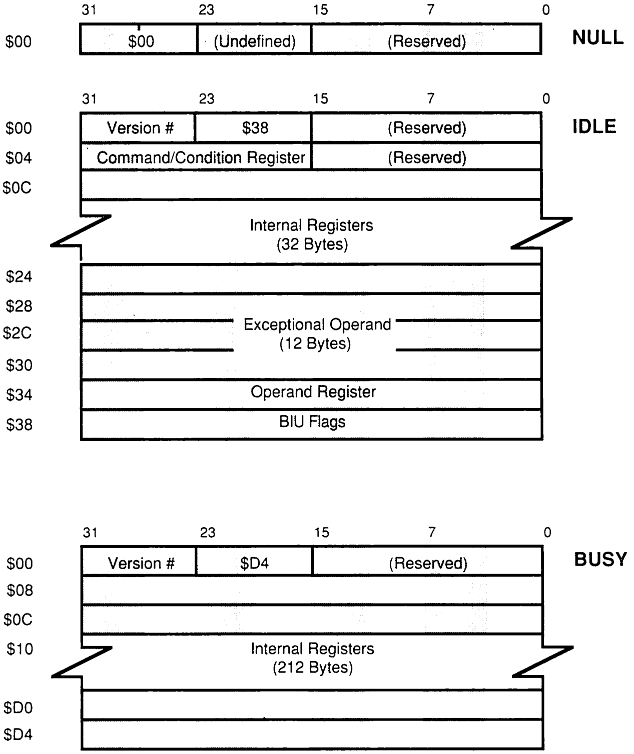

31
23
15
7
0
$00
$00
(Undefined)
(Reserved)
NULL
31
23
15
7
0
$00
Version #
$38
(Reserved)
IDLE
$04
Command/Condition Register
(Reserved)
$0C
Internal Registers
(32 Bytes)
$24
$28
$2C
Exceptional Operand
(12 Bytes)
$30
$34
Operand Register
$38
BIU Flags
31
23
15
7
0
$00
Version #
$D4
(Reserved)
BUSY
$08
$0C
$10
Internal Registers
(212 Bytes)
$D0
$D4

Figure 4-17. 68882 State Frames

family. Performing an FRESTORE using such a state frame sets all floating point data registers to non-signaling NANs and zeros the Floating Point Control Register (FPCR) and Floating Point Status Register (FPSR).

A hardware reset performs the equivalent of an FRESTORE of a NULL state frame. When you save the state of the coprocessor without having executed any floating point instructions, the coprocessor will use the NULL state frame. This is useful for multitasking systems, as explained below.

Idle State Frame The IDLE state is produced when the coprocessor is not currently executing an instruction or when the current instruction is close to completion (called the End Phase in Motorola documents). When the instruction is in the End Phase, it is shorter to wait for it to finish than to use a BUSY state frame. The main processor is made to wait in this situation until the instruction finishes and the IDLE format is used. You are guaranteed that the coprocessor is done with the instruction following completion of the FSAVE instruction in either case.

IDLE frames are useful in treating floating point exceptions. Items useful to the exception handler are the operand value that caused the exception and the Bus Interface Unit (BIU) flags quantity. The operand value can often be useful in determining the cause of the exception.

The BIU flags longword consists of a series of single bit flags. The low 16 bits of this word are currently reserved, and the upper 16 bits contain the information shown in Figure 4-18.

Bits 25–24 and 19–16 contain unpublished internal information and should not be modified. Bits 23–20 are a series of flags, one for each byte in the operand register field. If a bit in this field is set to one, the corresponding byte in the operand register field contains valid data.

Bit 26 indicates that the coprocessor has completed any necessary operand conversions and is ready to write an operand to memory. A zero in this bit indicates that a pending transfer has not been completed.

Bit 27 is perhaps the most important bit for an exception handler. If this bit is zero, it indicates that a floating point exception is pending. By taking the logical AND of the FPSR Exception Status Byte and the FPCR Exception Enable Byte (described previously), you can determine which exceptions are pending.

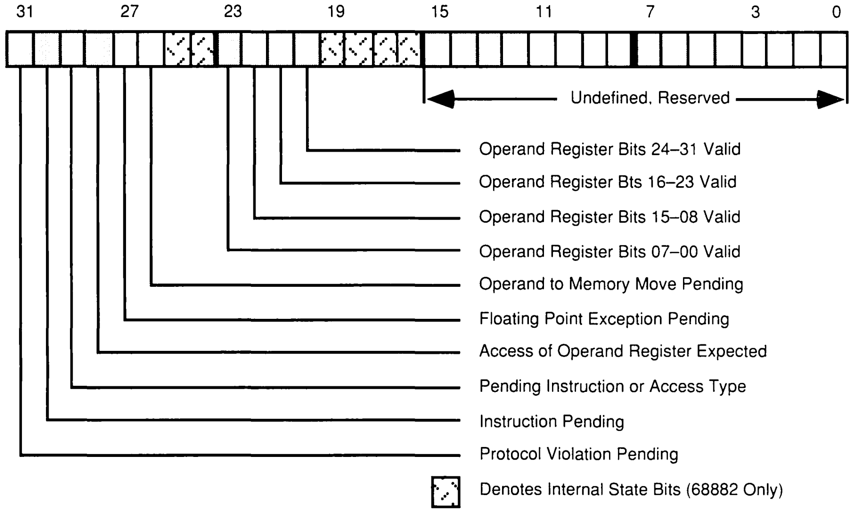

31
27
23
19
15
11
7
3
0
Undefined, Reserved
Operand Register Bits 24–31 Valid
Operand Register Bts 16–23 Valid
Operand Register Bits 15–08 Valid
Operand Register Bits 07–00 Valid
Operand to Memory Move Pending
Floating Point Exception Pending
Access of Operand Register Expected
Pending Instruction or Access Type
Instruction Pending
Protocol Violation Pending
Denotes Internal State Bits (68882 Only)

Figure 4-18. Bus Interface Unit (BIU) Flag Layout

If the exception handler needs to continue execution of the faulting program, it must set this bit before continuing. The sequence is to set the bit, do an FRESTORE, then perform an RTE. This will dismiss the exception and continue the faulting program. If you do not set the bit before the FRESTORE, the 68882 will take the exception again.

Another interesting use of this bit is to set an exception bit in the exception enable and exception status bytes and clear bit 27. If you perform an FRESTORE on the modified state frame, you can generate an arbitrary floating point exception.

Bits 30–28 determine what type of operation is pending (if any). Bit 28 indicates a pending access to the operand register. Bit 29 indicates the type of the access or a pending instruction. Bit 30 indicates a pending instruction. All of these bits are TRUE if they have the value zero. These bits may have the following values:

30	29	28	Meaning
0	0	0	(Undefined and Reserved)
0	0	1	Conditional Instruction Pending
0	1	0	(Undefined and Reserved)
0	1	1	General instruction Pending
1	0	0	Write of Operand CIR Pending
1	0	1	(Undefined and Reserved)
1	1	0	Read of Operand CIR Pending
1	1	1	No Pending Instruction or Operand CIR Access

The IDLE state can represent either a pending transfer of the operand Command Interface Register (CIR) or a pending instruction, as indicated above. Bits 30–28 should not be modified by the exception handler.

Bit 31 is a one if the coprocessor detected a coprocessor protocol violation.

BUSY State Frame The BUSY state frame indicates that a lengthy instruction is in process. Use of the BUSY format means that the time to save this rather large set of information is shorter than the time required to complete the instruction. Restoring this state frame will cause continuation of the instruction. You should not modify any information in this format state frame.

Floating Point Exceptions

The floating point coprocessor can generate eleven exceptions, as shown below:

Name	Vector	Address	Description
TRAPV	7	$1C	TRAPV (TRAPcc or FTRAPcc) instruction
Privilege Violation	8	$20	Executing a privileged instruction in user mode
Line F Emulator	11	$2C	Executing a word of the form $Fxxx
Coprocessor Protocol	13	$34	Hardware error in a coprocessor chip
Format Error	14	$38	Illegal state frame on FRESTORE
Unordered Condition	48	$C0	The FPSR BSUN bit was set by an instruction
Inexact Result	49	$C4	The INEX1 or INEX2 bits were set by an instruction
Division by Zero	50	$C8	Attempt to divide something by zero
Underflow	51	$CC	The UNFL bit was set by an instruction
Operand Error	52	$D0	The OPERR bit was set by an instruction
Overflow	53	$D4	The OVFL bit was set by an instruction
Not a Number (NAN)	54	$D8	The NAN bit was set by a signaling NAN operand

Exceptions through vectors 48 and higher can be masked by clearing the appropriate bit in the FPCR Exception Enable Byte. Other exceptions cannot be masked. Maskable exceptions use either a format 2 or format 9 stack frame, depending on whether the exception is recognized at the beginning of the instruction (format 0) or during instruction execution (format 9). An exception taken as the instruction begins is called a *pre-instruction exception.* An exception taken during execution is called a *mid-instruction exception.*

TRAPV (FTRAPcc) An exception through vector 7 is caused by execution of an FTRAPcc instruction in which the condition evaluates as TRUE. This exception uses a format 2 stack frame, as shown in Figure 2-39.

Privilege Violation Attempted execution of a privileged instruction while in user mode causes a privilege violation exception. The only privileged instructions in the floating point coprocessors are the FSAVE and FRESTORE instructions used for context switching. The privilege violation exception takes place through vector 8, using the default (format code 0) exception stack frame shown in Figure 2-34.

Line F Emulator Instructions that have $F as the upper 4 bits of the first word, but that are not valid coprocessor instructions, are reported as Line F exceptions to the main processor. The behavior is identical to that exhibited by machines without a coprocessor. Line F exceptions use a Format 0 (default) exception frame, as shown in Figure 2-34. The Line F vector is number 11. The stacked program counter value points to the instruction word causing the fault.

Coprocessor Protocol Whenever either the main processor or the coprocessor detects an improper response to the inter-chip dialog, a coprocessor protocol exception results. This exception takes place through vector 13, using a format 9 stack frame, shown in Figure 2-40. Unless you are performing the interprocessor protocol directly using the MOVES instruction (a topic beyond the scope of this book), this exception indicates a hardware problem.

Format Error When the coprocessor detects an invalid state frame on an FRESTORE instruction, a format error exception results. This exception uses a vector 14, with a format 0 (default) stack frame, as shown in Figure 2-34.

Unordered Condition (BSUN) What does it mean to compare a valid number to an invalid number? For instance, the quantity (0÷0) has no meaning mathematically. Is this number infinite or zero? How does it compare to a normal number? Whenever an instruction arrives at a result that is not meaningful, the 68881/2 chip places the NAN (Not A Number) quantity in the result operand and sets the NAN condition code bit in the FPSR. Any attempt to use an IEEE Non-Aware Condition in an instruction while the FPSR NAN bit is set raises the Unordered Condition exception.

This exception is indicated by the Branch or Set on UNordered Condition (BSUN) bit in the Exception Status byte of the Floating Point Status Register (FPSR). The condition is enabled by the bit of the same name in the Floating Point Control Register (FPCR) Exception Enable byte. If the exception is enabled, it uses vector 48, using either a format 0 or format 9 stack frame, as described above.

Inexact Result (INEX2/INEX1) Either the INEX1 or INEX2 exception bits in the FPSR may cause an inexact exception through vector 49. This exception uses either a format 0 or format 9 stack frame, as described above. INEX1 is generated when the source operand is in packed decimal number, and the value generates a repeating binary fraction (for example, 0.1 is not representable exactly in binary form). An INEX2 exception means that the precision chosen in the FPCR mode control byte is not large enough to contain the result of an operation. If the rounded result of an operation is not exact, the INEX2 exception is raised.

Both inexact exceptions share a single exception vector (49). The INEX bit in the FPSR Accumulated Exception byte indicates that one of these two conditions was raised since the last time INEX was cleared.

Zero Divide (DZ) Any attempt to divide something by zero raises the DZ exception flag in the FPSR. If the exception is enabled, a trap through vector 50 takes place. Note that this trap is different from the integer trap for zero divide, which uses vector 5. The floating Zero Divide exception can generate either a format 0 or format 9 stack frame, as discussed above.

Zero Divide can occur in the following instructions, given the input values noted below:

Instruction	Operand Values
FATANH	Source is ± 1
FDIV	Source is 0, and destination is not 0, NAN, or infinity
FLOG10	Source is 0
FLOG2	Source is 0
FLOGN	Source is 0
FLOGNP1	Source is -1
FSGLDIV	Source is 0, and destination is not 0, NAN, or infinity

If the exception is disabled, the destination receives the following results:

Instruction	Destination Value
FATANH	Plus infinity if source is -1; minus infinity if source is +1
FDIV	± infinity, sign determined by operand signs
FLOG10	Minus infinity
FLOG2	Minus infinity
FLOGN	Minus infinity
FLOGNP1	Minus infinity
FSGLDIV	± infinity, sign determined by operand signs

If the exception is enabled, the destination operand is not modified.

Underflow (UNFL) Underflow can result in one of two situations. The first is when an arithmetic result is too small to fit as a normalized number in a floating point data register (the exponent is less than or equal to the minimum value for the selected rounding precision.) Underflow can also occur when the number in a floating point register is too small to fit as a normalized number in an effective address destination using the selected precision. This means that the exponent is less than or equal to the minimum value for the selected destination data type. Note that for extended precision values, the exponent must be strictly less than the minimum value, due to the presence of the integer bit in this data format. Underflow cannot occur when the data type is byte, word, or long. The conversion simply stores zero without raising any exception condition. When the destination is packed decimal, both the underflow (UNFL) and operation error (OPERR) bits are set.

As is often the case, there is an exception to the rules. Both the FSGLMUL and FSGLDIV instructions do not generate overflow conditions unless the exponent is smaller than the value allowed for extended precision, rather than single precision.

The result actually stored in the destination operand depends on the rounding mode in effect. These values are as follows:

Rounding Mode	Final Value
Round to nearest	Zero, with the sign of the intermediate result
Round to zero	Zero, with the sign of the intermediate result
Round to - infinity	For positive underflow, +0 For negative underflow, smallest denormalized number
Round to +infinity	For positive underflow, smallest denormalized number For negative underflow, -0

Underflow exceptions use vector 51 and can generate either a format 0 or a format 9 stack frame, as previously discussed.

Operand Error (OPERR) Operand error (OPERR) exceptions are generated under a wide range of circumstances. An instruction generates an operand error whenever the function to be performed has no mathematical meaning for the value(s) specified. OPERR exceptions can be caused by the following instructions:

Instruction	Condition Causing OPERR
FACOS	Source is ±infinity, > +1, or < -1
FADD	(+infinity) + (-infinity), or (-infinity) + (+infinity)
FASIN	Source is ±infinity, > +1, or < -1
FATANH	Source is ±infinity, > +1, or < -1
FCOS	Source is ±infinity
FDIV	(0÷0) or (infinity÷infinity)
FGETEXP	Source is ±infinity
FGETMAN	Source is ±infinity
FLOG10	Source is < 0 or source is -infinity
FLOG2	Source is < 0 or source is -infinity
FLOGN	Source is < 0 or source is -infinity
FLOGNP1	Source is < -1 or source is -infinity
FMOD	Destination is ±infinity or source is 0 and destination is not NAN
FMOVE to B,W,L	Integer Over/Underflow, Source is NAN, or Source is ±infinity
FMOVE to P	Result exponent > 999 (decimal) or K-factor > 17 (FMOVE instruction)
FMUL	0 times ± infinity, or ±infinity times 0
FREM	Destination is ±infinity or source is 0 and destination is not NAN
FSCALE	Source is ±infinity and destination is not NAN
FSGLDIV	(0÷0) or (infinity÷infinity)
FSGLMUL	0 times ± infinity, or ±infinity times 0
FSIN	Source is ±infinity
FSINCOS	Source is ±infinity
FSQRT	Source is < 0 or source is -infinity
FSUB	Source and destination are both +infinity or -infinity
FTAN	Source is ±infinity

The value stored in the destination depends on the destination data type. An operand error is never generated when the destination effective address specifies an integer data register or memory and the format is S, D, or X. An error caused by integer overflow, or a floating point data register containing infinity will store the largest value (positive or negative) that can be accommodated by the destination data format. An integer destination with a source value of NAN will result in the most significant 8, 16, or 32 bits of the NAN mantissa being stored in the integer.

An operand error caused by an FMOVE packed decimal operation with a K-factor larger than 17 will behave as if the K-factor were 17 (K-factors are explained in the FMOVE instruction description). A packed decimal result with an exponent larger than 999 will cause an extra exponent digit to be stored in bits 79–76 of the result. If the destination is a floating point data register, it will receive an extended precision NAN value, with the mantissa set to all ones.

Operand Error exceptions use vector 52 and can generate either a format 0 or a format 9 stack frame, as discussed above.

Overflow (OVFL) The Overflow (OVFL) exception occurs when a destination value is too large to fit into the destination operand, using the selected rounding precision. Overflow occurs only when dealing with single, double, or extended precision destination operands. Converting a too-large value to integer or packed decimal formats signals an Operation Error (OPERR) instead of an overflow.

The FSGLDIV and FSGLMUL instructions are an anomaly, as is the Underflow (UNFL) exception. These instructions cause an overflow only when the exponent exceeds the maximum permitted value for extended precision, even though the instructions are for single precision only.

The result actually stored in the destination operand depends on the rounding mode in effect. These values are as follows:

Rounding Mode	Final Value
Round to nearest	Infinity, with the sign of the intermediate result
Round to zero	Largest magnitude number, with the sign of the intermediate result
Round to −infinity	For positive overflow, largest positive number For negative overflow, negative infinity
Round to +infinity	For positive overflow, positive infinity For negative overflow, largest negative number

Overflow exceptions use vector 53 and can generate either a format 0 or a format 9 stack frame, as discussed above.

Signaling NAN (SNAN) Whenever the source of a floating point operation is a signaling NAN, the SNAN exception is raised. If the exception is enabled, the processor takes an exception through vector 54 using a format 9 stack frame, as shown in Figure 2-40.

Multiple Exceptions Sometimes several exceptions can be signaled by a single instruction. In this case, the highest priority exception is taken, and the other conditions do not cause a trap. Exception priorities are

1. BSUN (Highest priority)
2. SNAN
3. OPERR
4. OVFL
5. UNFL
6. DZ
7. INEX2/INEX1 (Lowest priority)

Only certain combinations of exceptions can occur together. These are

1. SNAN and INEX1
2. OPERR and INEX2
3. OPERR and INEX1
4. OVFL and INEX2 and/or INEX1
5. UNFL and INEX2 and/or INEX1
6. INEX2 and INEX1

The trap handler must check for the presence of multiple exceptions in these cases.

Continuing From an Exception Some floating point exceptions can be corrected in software. The proper method for handling an exception is to perform an FSAVE instruction, adjust the state frame and any registers required, issue an FRESTORE on the (possibly modified) state frame, and execute an RTE instruction to continue the faulting program. The exception handler **must** set bit 27 of the BIU Flags field in an IDLE state frame before the FRESTORE. Otherwise, the exception will be repeated. Exceptions that can be continued must be either 68020/30 exceptions (like BUSERR, TRACE, etc.), or have an IDLE state frame. The exception handler should use only FMOVE, FSAVE, FRESTORE or main processor instructions in order to avoid getting another exception or changing the condition codes.

BSUN A BSUN (unordered) exception is the result of executing a floating conditional instruction with the NAN bit set in the FPSR condition code byte and with the BSUN bit set in the FPCR exception enable byte. Issuing an RTE will attempt to re-execute the failing conditional instruction. To prevent an infinite loop, you must ensure that the trap does not recur. Four techniques are available for this:

1. Clear the NAN bit in the FPSR condition code byte.
2. Disable the BSUN exception in the FPCR exception enable byte.
3. Set the condition codes to the desired value to cause the desired conditional execution.
4. Advance the program counter (on the stack) beyond the faulting instruction.

INEX1/2 An inexact result exception causes the destination to be stored, and the processor takes the exception. The address of the instruction generating the exception is in the FPIAR. If the destination is a memory location, the address is at offset $10 in the format 9 exception frame. Inexact exceptions differ from the others in that the exceptional operand field in the state frame is not defined. The trap handler must locate the operands by inspecting the faulting instruction. This is only the case if the inexact exception is the only one indicated. If multiple exceptions occur on a single instruction, the exceptional operand field is related to a higher-priority exception.

DZ Division by zero does not modify the destination operand. Use the FPIAR to determine the address of the failing instruction. Note that this exception can only occur on instructions where a floating point data register is the destination. Should you wish to change the value stored, you must do so manually.

UNFL An underflow exception stores the destination operand and then takes the exception. This can occur either on the instruction itself or on a subsequent floating point exception. Use the FPIAR to get the address of the faulting instruction. If the destination is a memory location, offset $10 of the exception stack frame (format 9) contains the address. For memory or main processor register destinations, the value stored as the exceptional operand in the state frame is the intermediate result mantissa, with a 15-bit exponent biased as a normal extended precision number. For floating point register destinations, the value in the exceptional operand field of the state frame is the intermediate result, rounded to extended precision. An additional exponent bias of +$6000 is used to allow the 17-bit intermediate exponent to be represented in 15 bits. (The exponent bias is $3FFF + $6000). To recover the original 17-bit quantity, take the 15-bit extended precision exponent, sign-extend to 17 bits (or more), and subtract the quantity $9FFF.

Attempting to calculate a number so small that it underflows the 17-bit intermediate exponent produces a condition called *catastrophic underflow*. The exponent value stored in this case is zero, which is the smallest possible value.

There are two conditions that cause the 68881/2 to signal an underflow. The first is that the value of the number is too small to represent as a normalized quantity in the target format. The second is that there is a loss of accuracy in calculating a very small number. The underflow (UNFL) flag in the accrued exception byte of the FPSR becomes set only when both these conditions are satisfied. The UNFL flag in the FPSR Exception Status byte is set whenever the result is too small to represent as a normalized number.

OPERR For Operand Error (OPERR) exceptions, the FPIAR contains the address of the offending instruction. The exception can occur with the program counter value in the exception stack frame pointing to a subsequent instruction. The FPIAR has the address of the instruction causing the fault. If the destination is an effective address, the coprocessor writes the results described in the OPERR section above. If the destination is a floating point data register, the destination has not been modified. The trap handler must overwrite the destination if desired. For an operand in memory, the address is contained in the exception stack frame (format 9) at offset $10.

OVFL An overflow exception stores the destination operand and then takes the exception. This can occur on the instruction itself or on a subsequent floating point exception. Use the FPIAR to get the address of the faulting instruction. If the destination is a memory location, offset $10 of the exception stack frame (format 9) contains the address. For a memory or main processor register destinations, the value stored as the exceptional operand in the state frame is the intermediate result mantissa rounded to the destination precision, with a 15-bit exponent biased as an extended precision number. For floating point data register destinations, the value in the exceptional operand field of the state frame is the intermediate result, rounded to extended precision. An additional exponent bias of -$6000 is used to allow the 17-bit intermediate exponent to be represented in 15 bits. (The exponent bias is $3FFF - $6000). To recover the original 17-bit quantity, take the 15-bit extended precision exponent, sign-extend to 17 bits (or more), and add the quantity $2001 (subtract $3FFF-$6000).

There is a special error condition that results from attempting to calculate a number so large that it overflows even the 17-bit intermediate value exponent. This condition is called *catastrophic overflow*, and the exponent value is set to zero in this case. This value cannot result from any other type of overflow.

SNAN You can use signaling NANs to implement custom data types in software. This technique involves taking an SNAN trap, performing the operation in software, and continuing the faulted instruction. You can overwrite the Exceptional Operand field in the state frame to simulate the result to be stored in the floating point data register. Since that FMOVE, FMOVEM, and FSAVE instructions cannot cause exceptions, you use these instructions to manipulate SNAN quantities.

Concurrency Considerations

As you read through the instruction descriptions, you will note that the vast majority of these instructions use a floating point data register as the destination operand. Effective address operands are usually allowed only as source operands. This is no accident. By structuring the coprocessor in this fashion, Motorola can allow the coprocessor to take the source operand, release the main processor, and perform the floating point calculation in parallel with main processor operations. The system must wait, of course, when the value (or condition codes) produced by the floating point instruction is required by another instruction.

Arithmetic operations come in two classifications: one source value (called *monadic*), or two source values (called *dyadic*). These operations are as follows:

Monadic Operations		Dyadic Operations	
Instruction	**Function**	**Instruction**	**Function**
FABS	Absolute Value	FADD	Add
FACOS	Arc cosine	FCMP	Compare
FASIN	Arc sine	FDIV	Divide
FATAN	Arc tangent	FMUL	Multiply
FCOS	Cosine	FREM	IEEE Remainder
FCOSH	Hyperbolic cosine	FSCALE	Scale Exponent
FETOX	e^x	FSGLDIV	Single precision divide
FETOXM1	$e^x - 1$	FSGLMUL	Single precision multiply
GETEXP	Get exponent	FSUB	Subtract
GETMAN	Get mantissa		
FINT	Get integer part		
FINTRZ	Integer, round to zero		
FLOGN	$\ln(x)$		
FLOGNP1	$\ln(x+1)$		
FLOG10	$\log_{10}(x)$		
FLOG2	$\log_2(x)$		
FNEG	Negate		
FSIN	Sine		
FSINH	Hyperbolic Sine		
FSQRT	Square Root		
FTAN	Tangent		
FTANH	Hyperbolic Tangent		
FTENTOX	10^x		
FTWOTOX	2^x		

These operations are typically compute-intensive and provide the rationale for concurrency.

68881 The 68881 provides limited concurrency. Some coprocessor and main processor instructions are overlapped. Fortunately, the coprocessor instructions that can be overlapped are those requiring the most execution time. When a second coprocessor instruction is initiated, both processors wait for the completion of the first instruction. By interleaving floating point and integer instructions, you can realize fairly significant performance improvements. (Matching the execution time of the floating point instructions would be ideal, but not realistic.)

68882 The 68882 is a bit more ambitious. It can overlap an arithmetic floating point operation with certain types of FMOVE instructions and certain types of arithmetic floating point operations. This is in addition to the main processor/coprocessor overlap described for the 68881. The instructions fall into four groups: no concurrency, minimum concurrency, partial concurrency, and full concurrency. The distinction is largely based on two factors: whether registers conflict, and whether the second instruction in the instruction stream requires a conversion from integer to floating point format.

NO CONCURRENCY A register conflict is a situation in which the destination of one instruction is either the source or destination operand of the next operand. The second instruction cannot proceed in parallel with the first if the second relies on the results from the first. The presence of a destination to source register conflict forces a no-concurrency situation. This type of conflict occurs when the source register of one instruction is the destination register of the previous instruction. The second instruction must wait until the first is completely finished.

Conditional instructions also exhibit no concurrent capability. Since the condition codes are usually the result of the previous floating point operation, overlapping these instructions is usually not possible. The 68882 does not allow a conditional instruction to execute concurrently with an instruction (like FMOVE) that does not affect the condition codes, though this is theoretically possible. FNOP is also included in this group to provide a means of ensuring that all floating point operations have completed before the main processor continues.

The 68882 has a single arithmetic processor, the APU. An instruction that simply moves data from one place to another (without conversion) can execute completely concurrently with another instruction, provided there is no register conflict. It is also possible to move one floating point format to another with complete concurrency.

MINIMAL CONCURRENCY Minimal concurrency involves one of the situations in the table below. The FMOVEM and FMOVECR instructions always force minimal concurrency. This condition also occurs whenever the APU must perform integer or packed conversion to floating point. An FMOVE instruction involving a control register means minimal concurrency as well.

Instruction	Operand Syntax	Operand Data Format
FMOVE	<ea>,FPn	B,W,L,P
	FPm,<ea>	B,W,L
	FPm,<ea>{#k}	P
	FPM,<ea>,Dn	P
	<ea>,FPcr	L
	FPcr,<ea>	L
FMOVECR	#ccc,FPn	X
FMOVEM	<ea>,<list>	L,X
	<ea>,Dn	X
	<list>,<ea>	L,X
	Dn,<ea>	X
FTST	FPm	B,W,L,P
F<aop>	<ea>,FPn	B,W,L,P
FSINCOS	<ea>,FPc:FPs	B,W,L,P

The instruction notation F<aop> denotes an arithmetic operation, either monadic or dyadic. For instructions requiring decimal conversion, the coprocessor can fetch any required source operands from memory, but must wait for the APU to convert them from decimal. The only instructions in this class that do not fall into this category are the FMOVECR and FMOVEM instructions, which can set the condition codes.

PARTIAL CONCURRENCY Instructions that can be somewhat overlapped are those that require no decimal conversion and which do not have register conflicts. These are

Instruction	Operand Syntax	Operand Data Format
FTST	<ea>	S,D,X
F<aop>	<ea>,FPn	S,D,X
	FPm,FPn	X
FSINCOS	<ea>,FPc:FPs	S,D,X
	FPm,FPc:FPs	X

The coprocessor can fetch memory operands or floating point data registers without requiring the APU. Register conflicts prevent any concurrency, however.

FULL CONCURRENCY For instructions to proceed completely in parallel, one instruction must not use the APU at all. The FMOVE instructions that meet this requirement are

Instruction	**Operand Syntax**	**Data Format**	**Notes**
FMOVE	FPm,FPn	S,D,X	1,2,3,6
FMOVE	<ea>,FPn	S,D,X	2,3,6
FMOVE	FPm,<ea>	S,D	1,2,4,5
FMOVE	FPm,<ea>	X	1,2

Notes:

1. No concurrency if FPm is also preceding instruction's destination register.
2. Partial concurrency if FPm contains NAN, Unnormalized, or Denormalized data.
3. Partial concurrency if Single or Double rounding mode set in FPCR.
4. Partial concurrency if INEX2 exception enabled in FPCR exception enable byte.
5. Partial concurrency if overflow or overflow occurs.
6. Partial concurrency if FPn conflicts with preceding instruction's destination register.

68882 PERFORMANCE OPTIMIZATION To maximize performance on the 68882, you should attempt to unroll loops, by using more registers. You should also perform two or more iterations of the loop before branching back to the loop top. It is also important to prevent register conflicts in this unrolled computation. For instance, given a loop of the form

```
FMOVE.x     (A0),FP1     ; Fetch Next Item
FADD.x      (A1),FP1     ; Add
FMOVE.x     FP1,(A2)     ; Store
```

You can provide optimization as follows:

```
FMOVE.x     (A0),FP1     ; Fetch Next
FMOVE.x     (A3),FP2     ; Fetch Next + 1
FADD.x      (A1),FP1     ; Add 1st
FADD.x          (A4),FP2        ; Add Next +1
FMOVE.x         FP1,(A2)        ; Store 1st
FMOVE.x         FP2,(A5)        ; Store Next + 1
```

This rather trivial example illustrates that careful movement of the instructions in an unrolled loop will avoid many register conflicts and APU contention. (The two FADDs in a row are only partially concurrent, for instance.)

Where possible, you should pair the fastest FMOVE instructions with the fastest arithmetic instructions, and the slowest FMOVE instructions with the slowest arithmetic instructions.

System Software Considerations

In addition to exception processing, context switching operations, interrupt handling, and coprocessor detection deserve special attention. Each is discussed below.

Context Switching Context switching is the art of suspending one program and reactivating another. The state of the machine, including all coprocessors, must be saved in memory for the suspended program and restored from memory for the program to be activated. There are three 68881/2 related data areas to be saved and restored: floating point data registers, floating point control registers, and the floating point state. You must be careful to save and restore these in the proper order so that the state will not be modified.

The appropriate technique for a context save is to perform an FSAVE of the coprocessor state, then use the FMOVEM instructions to save the floating point control registers (FPSR/FPCR/FPIAR) and floating point data registers. The total save operation is three instructions. Restoration is accomplished in the reverse order: Use FMOVEM to restore the data and control registers, then perform an FRESTORE on the saved coprocessor state. The order of control and data registers is not critical since the FMOVEM instruction for data registers does not affect the control registers.

In a multitasking system, you may wish to initialize the floating point state to a NULL state frame. When switching out of the task's context, you can inspect the state frame from the FSAVE instruction. If the state frame is a NULL frame, the task has not executed any floating point instructions. Thus, you need not save the control and data registers. This will reduce system overhead for systems with tasks that do no floating point instructions.

Interrupt Handling On a system with the 68882, an interrupt can occur between recognition of a floating point exception and the first instruction of the exception handler. Since there might be another exception in the pipeline, the interrupt handler must save the state of the coprocessor (as described for context switching) if it wishes to execute floating point instructions. However, the interrupt handler must **not** set bit 27 (exception pending) in the state frame before issuing the FRESTORE. This indicates to the coprocessor that the exception has not been satisfied and that execution of the faulting instruction will not be continued.

Coprocessor Detection Since the state frames and instruction timings are different between the 68881 and 68882, an operating system designer may wish to include code to determine the presence or absence of a floating point unit. It might also be desirable to determine whether the coprocessor is a 68881 or 68882. The technique for doing this is as follows:

1. Execute an FNOP instruction. If the CPU takes an F-line trap, you have no coprocessor.
2. Execute an FSAVE instruction.
3. If the frame size (offset 1 in the state frame) is $18, it is a 68881. Otherwise it is a 68882.

Instruction Description Format

The remainder of this chapter describes each coprocessor instruction in a format similar to that used in Chapter 3 for the main processor instructions. This section describes the layout of these descriptions. The 68881/2 coprocessors use the F-line instruction format described in Chapter 2. The Coprocessor ID field (bits 11–9 of the first word) of these instructions is 001 for Motorola-manufactured coprocessor chips. We will treat this field as a constant in this chapter.

Header

The header consists of the instruction name, mnemonic expansion, and assembly language syntax. For example, consider the header for the FABS instruction:

FABS Instruction	**Syntax:**	**FABS.f**	**<ea>,FPn**
Floating Absolute Value		**FABS.X**	**FPm,FPn**
		FABS.X	**FPn**

The notation inst.f indicates the source data format. These formats are listed in the section on Data Formats at the beginning of this chapter.

Description

The next section of the explanation contains a prose description of the instruction, as well as a chart with the possible exceptions and allowed data formats, as shown below:

Exceptions								**Data Formats**						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	0	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Cleared.
UNFL	Set if operand is a denormalized extended precision number. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Cleared.

This example is also from the FABS instruction. The exceptions are those that occur in the FPCR exception enable byte and FPSR exception status byte. An exception must be enabled to actually occur. This description details which exceptions will be signaled in the FPSR exception status byte, regardless of whether they will actually trap. The data formats are those listed in the Data Formats section, corresponding to those allowed on the instruction in the description header above.

Numerical Results

Many floating point functions have meaning only for certain ranges of values. These instructions (logarithms, some transcendental functions and so on), have a Numerical Results section, with a chart of the following form:

Destination		Source					
		In Range		Zero		Infinity	
		+	−	+	−	+	−
In Range	+	Add		Add		+infinity	−infinity
	−						
Zero	+	Add		+ 0.0	0.0	+infinity	−infinity
	−			0.0	−0.0		
Infinity	+	+infinity		+ infinity		+infinity	NAN(OPERR)
	−	− infinity		−infinity		NAN(OPERR)	−infinity

This chart has allowable ranges for both source and destination for operations that have both source and destination values (*dyadic*). Note that the results sometimes depend on the signs of both operands. If an operation has only a source value (*monadic*), the destination form of the chart has only one set of values. (This example is from the FADD instruction.)

Addressing Modes

Most floating point operations have the effective address field as the source, and thus have only one permissible field rather than two as in Chapter 3. We use a figure of the following form for those instructions:

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long or single data types

A few instructions can have either source or destination operands. These instructions use a figure with the same format as those in Chapter 3.

Instruction Format

As in Chapter 3, a figure shows the layout of the bits in the instruction words, as shown below:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td colspan="3">Mode</td><td colspan="3">Register</td></tr>
<tr><td>0</td><td>R/M</td><td>0</td><td colspan="3">Source Spec</td><td colspan="3">Dest Register</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
</table>

This is the layout of the FABS instruction. As noted above, bits 11–9 (the coprocessor ID) are always 001 in this book. Each of the fields is described in a prose description immediately following the figure.

Example

The last part of an instruction description is a sample instruction. The instruction is shown in a figure like the one above. The debugger is used to show the instruction execution.

For the instruction FRESTORE (A0), the diagram is

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	1	Mode			Register		
1	1	1	1	0	0	1	1	0	1	0	1	0	0	0	0

The hex value is $F350. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8b28 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 0000fff0 BSUN SNAN OPERR OVFL UNFL DZ INEX2 INEX1 Rsv Round +inf
FPSR: 0ffffff8 N Z I NAN ESTAT BSUN SNAN OPERR OVFL UNFL DZ INEX2 INEX1
  Accrued: IOP OVFL UNFL DZ INEX FPIAR: ffffffff
FP0: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP2: 4000 8000000000000000 Decimal: +2.0000000000000000E+000
FP3: 4000 c000000000000000 Decimal: +3.0000000000000000E+000
FP4: 4001 8000000000000000 Decimal: +4.0000000000000000E+000
FP5: 4001 a000000000000000 Decimal: +5.0000000000000000E+000
FP6: 4001 c000000000000000 Decimal: +6.0000000000000000E+000
FP7: 4001 e000000000000000 Decimal: +7.0000000000000000E+000
PC =FRESTORE$        FRESTORE      (A0)
```

The memory area addressed by A0 contains the following data:

```
b03f8b28:        00000000
```

Note that this is a null state frame, which is four bytes long. Restoring this frame yields the following:

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8b28 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
  Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 7fff ffffffffffffffff Decimal: +0.ffffffffffffffffE-fff
FP1: 7fff ffffffffffffffff Decimal: +0.ffffffffffffffffE-fff
FP2: 7fff ffffffffffffffff Decimal: +0.ffffffffffffffffE-fff
FP3: 7fff ffffffffffffffff Decimal: +0.ffffffffffffffffE-fff
FP4: 7fff ffffffffffffffff Decimal: +0.ffffffffffffffffE-fff
FP5: 7fff ffffffffffffffff Decimal: +0.ffffffffffffffffE-fff
FP6: 7fff ffffffffffffffff Decimal: +0.ffffffffffffffffE-fff
FP7: 7fff ffffffffffffffff Decimal: +0.ffffffffffffffffE-fff
PC =FRESTORE$+002   FNOP
```

This example illustrates the extra fields required to debug floating point instructions. Both the floating point control registers and the floating point data registers are shown before and after. Most instruction descriptions do not alter more than one data register, and only those registers required are shown. In this example, all the information has been altered. Note that the fields of the floating point control register and floating point control register are shown both in hex and in mnemonic form. As in Chapter 3, values modified by the instruction are underlined in the **After** display.

The actual instruction sets for this chapter start on the next page.

FABS Instruction

Floating Absolute Value

Syntax: FABS.f <ea>,FPn
FABS.X FPm,FPn
FABS.X FPn

Description

The FABS instruction takes the absolute value of a floating point number and places the result in a floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	0	*	0	*	0	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Cleared.
UNFL	Set if operand is a denormalized extended precision number. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Cleared.

Numerical Results

	Source		
Destination	In Range + −	Zero + −	Infinity + −
Result	Absolute Value	Absolute Value	Absolute Value

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An	100	An	•
(d16,An	101	An	•
(d8,An,Xn	110	An	•
(bd,An,Xn	110	An	•
([bd,An,Xn],od	110	An	•
([bd,An],Xn,od	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC	111	010	•
(d8,PC,Xn	111	011	◦
(bd,PC,Xn)	111	011	◦
([bd,PC,Xn],od	111	011	•
([bd,PC],Xn,od	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	0	0	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FABS FP0,FP1 instruction places the absolute value of floating point data register 0 in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	0	0	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0

The hex value is $F2000098. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 08000000 N - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: c002 9000000000000000 Decimal: -9.0000000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FABS$       FABS       FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
  Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: c002 9000000000000000 Decimal: -9.0000000000000000E+000
FP1: 4002 9000000000000000 Decimal: +9.0000000000000000E+000
PC =FABS$+004   FNOP
```

FACOS Instruction

Floating Arc Cosine function

Syntax: FACOS.f <ea>,FPn
FACOS.X FPm,FPn
FACOS.X FPn

Description

The FACOS instruction takes the arc cosine of a floating point number and places the result in a floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	0	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source is infinity, > +1, or < -1. Cleared otherwise.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Arc Cosine		+ π/2		NAN (OPERR)	

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An		xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long ,or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	1	0	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FACOS FP0,FP1 instruction places the arc cosine of the value in floating point data register 0 in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	1	0	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0

The hex value is $F200009C. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FACOS$      FACOS      FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
FP1: 3fff c90fdaa22168c235 Decimal: +1.5707963267948966E+000
PC =FACOS$+004  FNOP
```

The arc cosine of zero (in FP0) is π/2, the value in FP1 following the instruction.

FADD Instruction

Floating Add

Syntax: **FADD.f** **<ea>,FPn**
FADD.X **FPm,FPn**

Description

The FADD instruction adds an operand to a floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	*	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if adding +infinity to −infinity. Cleared otherwise.
OVFL	Set if overflow occurred. Cleared otherwise.
UNFL	Set if underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination		Source					
		In Range		Zero		Infinity	
		+	−	+	−	+	−
In Range	+	Add		Add		+infinity	− infinity
	−						
Zero	+	Add		+ 0.0	0.0	+infinity	− infinity
	−			0.0	−0.0		
Infinity	+	+infinity		+infinity		+infinity	NAN(OPERR)
	−	−infinity		− infinity		NAN(OPERR)	− infinity

Exception bits that are always set by a combination of operand ranges are indicated in parentheses. Adding +zero to +zero or −zero to −zero produces +zero in RN, RZ, and RP rounding modes, and −zero in RM mode.

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An		xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	0	1	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FADD FP0,FP1 instruction places the sum of floating point registers FP0 and FP1 in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	0	1	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0

The hex value is $F20000A2. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP1: 4000 8000000000000000 Decimal: +2.0000000000000000E+000
PC =FADD$       FADD       FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP1: 4000 c000000000000000 Decimal: +3.0000000000000000E+000
PC =FADD$+004   FNOP
```

FASIN Instruction

Floating Arc Sine function

Syntax: FASIN.f <ea>,FPn
FASIN.X FPm,FPn
FASIN.X FPn

Description

The FASIN instruction takes the arc sine of a floating point number and places the result in a floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	0	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source is infinity, > +1, or < −1. Cleared otherwise.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Arc Sine		+0.0	−0.0	NAN (OPERR)	

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An		xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	1	0	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	**Suffix**	**Data Type**
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FASIN FP0,FP1 instruction places the arc sine of the value in floating point data register 0 in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	1	0	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0

The hex value is $F200008C. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FASIN$      FASIN      FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP1: 3fff c90fdaa22168c235 Decimal: +1.5707963267948966E+000
PC =FASIN$+004  FNOP
```

The arc sine of 1 (in FP0) is $\pi/2$, the value in FP1 following the instruction.

FATAN Instruction

Floating Arc Tangent function

Syntax: **FATAN.f <ea>,FPn**
FATAN.X FPm,FPn
FATAN.X FPn

Description

The FATAN instruction takes the arc tangent of a floating point number and places the result in a floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	0	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Cleared.
UNFL	Set if underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Arc Tangent		+0.0	−0.0	+ π/2	−π/2

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An		xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	0	1	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FATAN FP0,FP1 instruction places the arc tangent of the value in floating point data register 0 in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	0	1	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0

The hex value is $F200008A. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FATAN$      FATAN      FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000C00a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP1: 3ffe c90fdaa22168c241 Decimal: +7.8539816339744830E-001
PC =FATAN$+004  FNOP
```

The arc tangent of 1 (in FP0) is the value in FP1 following the instruction.

FATANH Instruction

Floating Hyperbolic Arc Tangent function

Syntax: FATANH.f <ea>,FPn
FATANH.X FPm,FPn
FATANH.X FPn

Description

The FATANH instruction takes the hyperbolic arc tangent of a floating point number and places the result in a floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	*	*	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source operand is greater than +1 or less than -1. Cleared otherwise.
OVFL	Cleared.
UNFL	Set if underflow occurred. Cleared otherwise.
DZ	Set if source operand is equal to +1 or -1. Cleared otherwise.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Hyperbolic Arc Tangent		+0.0	− 0.0	NAN (OPERR)	

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An		xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	1	0	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	**Suffix**	**Data Type**
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FATANH FP0,FP1 instruction places the hyperbolic arc tangent of the value in floating point data register 0 in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	1	0	1

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1

The hex value is $F200008D. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FATANH$     FATANH     FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
FP1: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
PC =FATANH$+004            FNOP
```

The hyperbolic arc tangent of 0 (in FP0) is 0 (the value in FP1 following the instruction).

FBcc Instruction

Branch on Floating Point Condition

Syntax **FBcc.W** **<label>**
FBcc.L **<label>**

Description

The FBcc instruction evaluates a floating point condition combination and takes a branch if that condition is true. This instruction is similar to the Bcc main processor instruction, except that the coprocessor evaluates the condition.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
*	–	–	–	–	–	–	–		•	•				

BSUN	Set on some conditional tests (see below) if the NAN bit is set. Cleared otherwise.
SNAN	Not affected.
OPERR	Not affected.
OVFL	Not affected.
UNFL	Not affected.
DZ	Not affected.
INEX1	Not affected.
INEX2	Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	Size	Conditional Predicate					
16-bit displacement if Size = 0															
32-bit displacement if Size = 1															

The size field (bit 6 of the first instruction word) is zero for a 16-bit offset and 1 for a 32-bit offset. The PC value used in computing the offset is the address of the second word of the instruction.

The Conditional Predicate contains the encoding for the coprocessor condition to be interrogated. These combinations are as follows:

Mnemonic	Mnemonic Expansion	Equation	Predicate
F	False	0	000000
EQ	Equal	Z	000001
OGT	Ordered Greater Than	~(NAN | Z | N)	000010
OGE	Ordered Greater Than or Equal	Z | ~(NAN | N)	000011
OLT	Ordered Less Than	N & ~(NAN | Z)	000100
OLE	Unordered or Less Than or Equal	Z | (N & ~NAN)	000101
OGL	Ordered Greater or Less Than	~(NAN | Z)	000110
OR	Ordered	~NAN	000111
UN	Unordered	NAN	001000
UEQ	Unordered or Equal	NAN | Z	001001
UGT	Unordered or Greater Than	NAN | ~(N | Z)	001010
UGE	Unordered or Greater Than or Equal	NAN | Z | N	001011
ULT	Unordered or Less Than	NAN | (N & ~Z)	001100
ULE	Unordered or Less Than or Equal	NAN | Z | N	001101
NE	Not Equal	~Z	001110
T	True	1	001111
SF	Signaling False	0	010000
SEQ	Signaling Equal	Z	010001
GT	Greater Than	~(NAN | Z | N)	010010
GE	Greater Than or Equal	Z | ~(NAN | N)	010011
LT	Less Than	N & ~(NAN | Z)	010100
LE	Less Than or Equal	Z | (N& ~NAN)	010101
GL	Greater or Less Than	~(NAN | Z)	010110
GLE	Greater, Less, or Equal	~NAN	010111
NGLE	Not (Greater, Less, or Equal)	NAN	011000
NGL	Not(Greater or Less Than)	NAN | Z	011001
NLE	Not (Less Than or Equal)	NAN | ~(N | Z)	011010
NLT	Not Less Than	NAN | (Z | ~N)	011011
NGE	Not (Greater Than or Equal)	NAN | (N & ~Z)	011100
NGT	Not Greater Than	NAN | Z | N	011101
SNE	Signaling Not Equal	~Z	011110
ST	Signaling True	1	011111

▭ Can set Status Register BSUN exception bit

The shaded conditions set the BSUN (Branch / Set on UNordered) bit in the FPSR if the NAN condition code is set. The conditions are explained in detail in the Architecture description earlier in this chapter.

Example The FBLT *+$0010 instruction branches to the instruction at the address of the branch plus 10 hex if the floating point N bit is set and the NAN and Z bits are both clear. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	Size	Conditional Predicate					
16–bit displacement if Size = 0															
32–bit displacement if Size = 1															

1	1	1	1	0	0	1	0	1	0	0	1	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

The hex value is $F294000E. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 08000000 N - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
PC =FBCC$      FBLT       *+$0010      ; FBCC$+010
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 08000000 N - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
PC =FBCC$+010  FNOP
```

FCMP Instruction

Floating Compare

Syntax: FCMP.f <ea>,FPn
FCMP.X FPm,FPn

Description

The FCMP instruction performs a floating point comparison that is similar to the CMP instruction for integer data types. The floating point condition codes are set as a result of subtracting the source operation from a floating point data register. The source operand can be either an effective address operand or a floating point data register. Neither operand is modified.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	0	0	0	*	0	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Cleared.

Condition Codes

Destination		Source					
		In Range		Zero		Infinity	
		+	−	+	−	+	−
In Range	+	N,Z	none	none	none	N	none
	−	N	N,Z	N	N	N	none
Zero	+	N	none	Z	Z	N	none
	−	N	none	N,Z	N,Z	N	none
Infinity	+	none	none	none	none	Z	none
	−	N	N	N	N	N	N,Z

Underlined condition codes are always set when the operands are in the range in question. Condition codes shown in plain text are either set or cleared, depending on the operand values. The infinity bit is always cleared, and the NAN bit is set if either operand is a NAN.

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An		xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	1	1	0	0	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real

011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FCMP FP0,FP1 instruction compares the value in floating point data register 0 with the value in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	1	1	0	0	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	0

The hex value is $F20000B8. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4002 b000000000000000 Decimal: +1.1000000000000000E+001
FP1: 4002 a000000000000000 Decimal: +1.0000000000000000E+001
PC =FCMP$          FCMP   FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 08000000 N - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4002 b000000000000000 Decimal: +1.1000000000000000E+001
FP1: 4002 a000000000000000 Decimal: +1.0000000000000000E+001
PC =FCMP$+004      FNOP
```

This example compares the value 10 (in FP0) with 11 (in FP1). The N condition code (in FPSR) is set as a result.

FCOS Instruction

Floating Cosine function

Syntax:
FCOS.f <ea>,FPn
FCOS.X FPm,FPn
FCOS.X FPn

Description

The FCOS instruction takes the cosine of a floating point number and places the result in a floating point data register. The source is converted to extended precision (if required) before use. This function is not defined if the source is +infinity or −infinity. If the source is not in the range [-2π to $+2\pi$], it is reduced to that range. Large arguments may lose accuracy in this reduction process; arguments greater than approximately 10^{20} may lose all accuracy.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	0	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source operand is + infinity or −infinity. Cleared otherwise.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Cosine		+ 1.0		NAN (OPERR)	

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	1	0	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register field (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FCOS FP0,FP1 instruction places the cosine of the value in floating point data register 0 in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	1	0	1

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	1

The hex value is $F200009D. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
FP1: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
PC =FCOS$       FCOS       FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FCOS$+004   FNOP
```

The cosine of zero (in FP0) is 1.0, the value in FP1 following the instruction.

FCOSH Instruction

Floating Hyperbolic Cosine function

Syntax: **FCOSH.f <ea>,FPn**
FCOSH.X FPm,FPn
FCOSH.X FPn

Description

The FCOSH instruction takes the hyperbolic cosine of a floating point number and places the result in a floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	*	0	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Set if an overflow occurred. Cleared otherwise.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source		
	In Range + −	Zero + −	Infinity + −
Result	Hyperbolic Cosine	+ 1.0	+Infinity

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An		xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long , or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	0	0	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is:

Value	**Suffix**	**Data Type**
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FCOSH FP0,FP1 instruction places the hyperbolic cosine of the value in floating point data register 0 in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	0	0	1

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	1

The hex value is $F2000099. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
FP1: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
PC =FCOSH$     FCOSH     FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FCOSH$+004  FNOP
```

The hyperbolic cosine of zero (in FP0) is 1.0, the value in FP1 following the instruction.

FDBcc Instruction

Syntax: FDBcc Dn,<label>

Test Floating Point Condition, Decrement, and Branch

Description

The FDBcc instruction evaluates a floating point condition combination. If the specified condition is false, the low word of the data register is decremented. If the resulting value is not -1, a branch to the specified label is taken. FDBcc is normally used as the last instruction in a loop. Note that an initial value of -1 in the data register will cause 65,536 iterations (the largest possible value using FDBcc) of the loop. The condition may be viewed as the termination condition of a loop, much like the do/while construct in C or the REPEAT/UNTIL construct in the Pascal language. The FDBcc condition, unlike the language constructs, terminates the loop when the condition is true rather than when false. This instruction is similar to the DBcc main processor instruction, except that the coprocessor evaluates the condition.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
*	–	–	–	–	–	–	–		•	•				

BSUN	Set on some conditional tests (see below) if the NAN bit is set. Cleared otherwise.
SNAN	Not affected.
OPERR	Not affected.
OVFL	Not affected.
UNFL	Not affected.
DZ	Not affected.
INEX1	Not affected.
INEX2	Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	1	0	0	1	DataRegister		
0	0	0	0	0	0	0	0	0	0	Conditional Predicate					
16-bit displacement															

Mnemonic	Mnemonic Expansion	Equation	Predicate
F	False	0	000000
EQ	Equal	Z	000001
OGT	Ordered Greater Than	~(NAN | Z | N)	000010
OGE	Ordered Greater Than or Equal	Z | ~(NAN | N)	000011
OLT	Ordered Less Than	N & ~(NAN | Z)	000100
OLE	Unordered or Less Than or Equal	Z | (N & ~NAN)	000101
OGL	Ordered Greater or Less Than	~(NAN | Z)	000110
OR	Ordered	~NAN	000111
UN	Unordered	NAN	001000
UEQ	Unordered or Equal	NAN | Z	001001
UGT	Unordered or Greater Than	NAN | ~(N | Z)	001010
UGE	Unordered or Greater Than or Equal	NAN | Z | N	001011
ULT	Unordered or Less Than	NAN | (N & ~Z)	001100
ULE	Unordered or Less Than or Equal	NAN | Z | N	001101
NE	Not Equal	~Z	001110
T	True	1	001111
SF	Signaling False	0	010000
SEQ	Signaling Equal	Z	010001
GT	Greater Than	~(NAN | Z | N)	010010
GE	Greater Than or Equal	Z | ~(NAN | N)	010011
LT	Less Than	N & ~(NAN | Z)	010100
LE	Less Than or Equal	Z | (N& ~NAN)	010101
GL	Greater or Less Than	~(NAN | Z)	010110
GLE	Greater, Less, or Equal	~NAN	010111
NGLE	Not (Greater, Less, or Equal)	NAN	011000
NGL	Not(Greater or Less Than)	NAN | Z	011001
NLE	Not (Less Than or Equal)	NAN | ~(N | Z)	011010
NLT	Not Less Than	NAN | (Z | ~N)	011011
NGE	Not (Greater Than or Equal)	NAN | (N & ~Z)	011100
NGT	Not Greater Than	NAN | Z | N	011101
SNE	Signaling Not Equal	~Z	011110
ST	Signaling True	1	011111

▭ Can set Status Register BSUN exception bit

The Conditional Predicate contains the encoding for the coprocessor condition to be interrogated. These combinations are as follows:

The shaded conditions set the BSUN (Branch / Set on UNordered) bit in the FPSR if the NAN condition code is set. The conditions are explained in detail in the Architecture description earlier in this chapter.

Example The FDBEQ D0,*-$0020 instruction continues to the next instruction if the Z bit in the Floating Point Status Register (FPSR) is set. Otherwise, D0.W is decremented. If the result is not − 1, the branch is taken. This instruction has the following layout:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td colspan="3">DataRegister</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="6">Conditional Predicate</td></tr>
<tr><td colspan="16">16-bit displacement</td></tr>
</table>

1	1	1	1	0	0	1	0	0	1	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0

The hex value is $F248 0001 FFDC. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 00000001 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
PC =FDBCC$     FDBEQ      D0,*-$0020   ; FDBDST$
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 00000001 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
PC =FDBCC$+006  FNOP
```

Since the Z bit was set, the FDBEQ instruction did not branch.

FDIV Instruction

Floating Point Divide

Syntax: FDIV.f <ea>,FPn
FDIV.X FPm,FPn

Description

The FDIV instruction divides the destination floating point data register by the source operand. The result is left in the destination floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	*	*	*	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set for 0 ÷ 0 or infinity ÷ infinity. Cleared otherwise.
OVFL	Set if an overflow occurred. Cleared otherwise.
UNFL	Set if an underflow occurred. Cleared otherwise.
DZ	Set if source operand is zero and destination is in range. Cleared otherwise.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination		Source					
		In Range		Zero		Infinity	
		+	−	+	−	+	−
In Range	+	Divide		+infinity (DZ)	−nfinity	+0.0	−0.0
	−			−infinity	+ infinity	−0.0	+0.0
Zero	+	+ 0.0	−0.0	NAN (OPERR)		+ 0.0	−0.0
	−	− 0.0	+0.0			−0.0	+0.0
Infinity	+	+ infinity	−infinity	+infinity	−infinity	NAN(OPERR)	
	−	− infinity	+infinity	−infinity	+infinity		

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	0	0	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FDIV FP0,FP1 instruction places the quantity (FP1 ÷ FP0) in floating point data register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	0	0	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0

The hex value is $F20000A0. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 00000001 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4001 e000000000000000 Decimal: +7.0000000000000000E+000
FP1: 4003 b000000000000000 Decimal: +2.2000000000000000E+001
PC =FDIV$       FDIV       FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 00000001 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 4001 e000000000000000 Decimal: +7.0000000000000000E+000
FP1: 4000 c924924924924925 Decimal: +3.1428571428571429E+000
PC =FDIV$+004   FNOP
```

This example divides 22 by 7, with the result in FP1 following the instruction.

FETOX Instruction

e^x Function

Syntax: FETOX.f <ea>,FPn
FETOX.X FPm,FPn
FETOX.X FPn,FPn

Description

The FETOX instruction raises the quantity e to the power indicated by the source operand. The result is left in the destination floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	*	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Set if an overflow occurred. Cleared otherwise.
UNFL	Set if an underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	e^x		+ 1.0		+infinity	+0.0

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	0	0	0	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FETOX FP0,FP1 instruction places the quantity $e^{(FP0)}$ (the exponent is contained in FP0) in floating point data register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	0	0	0	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0

The hex value is $F2000090. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FETOX$      FETOX      FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP1: 4000 adf85458a2bb4a8c Decimal: +2.7182818284590452E+000
PC =FETOX$+004  FNOP
```

This example raises e to the first power, giving the quantity e in FP1 following the instruction.

FETOXM1 Instruction

e^x-1 Function

Syntax:

FETOXM1.f	**<ea>,FPn**
FETOXM1.X	**FPm,FPn**
FETOXM1.X	**FPn,FPn**

Description

The FETOXM1 instruction raises the quantity e to the power indicated by the source operand, then subtracts one from that value. The result is left in the destination floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	*	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Set if an overflow occurred. Cleared otherwise.
UNFL	Set if an underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	$e^x - 1$		+ 0.0	−0.0	+infinity	+0.0

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	0	0	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	**Suffix**	**Data Type**
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FETOXM1 FP0,FP1 instruction places the quantity $e^{(FP0)}-1$ (the exponent is contained in FP0) in floating point data register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	0	0	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0

The hex value is $F2000088. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FETOXM1$    FETOXM1    FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP1: 3fff dbf0a8b145769519 Decimal: +1.7182818284590452E+000
PC =FETOXM1$+004           FNOP
```

The exponent is 1.0, giving the quantity e-1 in FP1 following the instruction.

FGETEXP Instruction

Get Exponent

Syntax: FGETEXP.f <ea>,FPn
FGETEXP.X FPm,FPn
FGETEXP.X FPn,FPn

Description

The FGETEXP instruction converts the source operand to extended precision (if necessary) and extracts the binary exponent. The exponent bias is removed, and the resulting quantity is converted to an extended precision number and placed in the destination floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	0	0	0	*	0	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Cleared.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Exponent		+ 0.0	− 0.0	NAN (OPERR)	

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	1	1	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	**Suffix**	**Data Type**
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register field (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FGETEXP FP0,FP1 instruction places the exponent from floating point data register 0 in floating point data register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	1	1	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	0

The hex value is $F200009E. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 400b 8000000000000000 Decimal: +4.0960000000000000E+003
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FGETEXP$    FGETEXP    FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 400b 8000000000000000 Decimal: +4.0960000000000000E+003
FP1: 4002 c000000000000000 Decimal: +1.2000000000000000E+001
PC =FGETEXP$+004           FNOP
```

The original quantity in FP0 is 4096, or 2^{12}. The binary exponent is therefore 12 (in FP1 following the instruction).

FGETMAN Instruction

Get Mantissa

Syntax: FGETMAN.f <ea>,FPn
FGETMAN.X FPm,FPn
FGETMAN.X FPn,FPn

Description

The FGETMAN instruction converts the source operand to extended precision (if necessary) and extracts the mantissa. The resulting quantity is converted to an extended precision number and placed in the destination floating point data register. This result can be in the range 1.0 to 2.0 (inclusive), zero, or a NAN. The sign of the result is the same as the sign of the source mantissa.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	0	0	0	*	0	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Cleared.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Mantissa		+0.0	−0.0	NAN (OPERR)	

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	1	1	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FGETMAN FP0,FP1 instruction places the mantissa from floating point data register 0 in floating point data register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	1	1	1

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1

The hex value is $F200009F. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4016 bc614e0000000000 Decimal: +1.2345678000000000E+007
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FGETMAN$    FGETMAN    FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4016 bc614e0000000000 Decimal: +1.2345678000000000E+007
FP1: 3fff bc614e0000000000 Decimal: +1.4717195034027100E+000
PC =FGETMAN$+004 FNOP
```

The original quantity in FP0 is 12345678. Notice that although the mantissa is the same in the result, the difference in exponents causes the decimal result to be radically different.

FINT Instruction

Get Integer Part

Syntax: **FINT.f <ea>,FPn**
FINT.X FPm,FPn
FINT.X FPn,FPn

Description

The FINT instruction converts the source operand to extended precision (if necessary) and extracts the integer part. The resulting quantity is converted to an extended precision number and placed in the destination floating point data register. The sign of the result is the same as the sign of the source mantissa.

Results of this instruction are affected by the current rounding mode in the Floating Point Control Register (FPCR). The quantity 1.51 rounds to 2.0 for the round to nearest and round to plus infinity modes and 1.0 for the round to zero and round to minus infinity modes. A related instruction, FINTRZ, always rounds to zero regardless of the setting of the FPCR.

Exceptions								**Data Formats**						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	0	0	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if an inexact conversion occurred. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Integer		+ 0.0	− 0.0	+infinity	− infinity

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	0	0	0	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	**Suffix**	**Data Type**
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FINT FP0,FP1 instruction places the integer part of floating point data register 0 in floating point data register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	0	0	0	1

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1

The hex value is $F2000081. Executing the instruction yields:

Before:

```
USP=b03f838c MSP=b03f838c ISP =b03f838a SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f838a
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 3fff c147ae147ae147ae Decimal: +1.5100000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FINT$       FINT       FP0,FP1
```

After:

```
USP=b03f838c MSP=b03f838c ISP =b03f838a SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f838a
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 3fff c147ae147ae147ae Decimal: +1.5100000000000000E+000
FP1: 4000 8000000000000000 Decimal: +2.0000000000000000E+000
PC =FINT$+004   FNOP
```

Since the Round to Nearest mode was in effect, 1.51 rounds to 2.0 (in FP1 following the instruction).

FINTRZ Instruction

Get Integer Part, Round to Zero

Syntax: **FINTRZ.f <ea>,FPn**
FINTRZ.X FPm,FPn
FINTRZ.X FPn,FPn

Description

The FINTRZ instruction converts the source operand to extended precision (if necessary) and extracts the integer part. The resulting quantity is converted to an extended precision number and placed in the destination floating point data register. The sign of the result is the same as the sign of the source mantissa. Results of this instruction are unaffected by the current rounding mode in the Floating Point Control Register (FPCR). Results are always rounded to zero (sometimes called *chop mode*). The quantity 1.51 always rounds to 1.0. This instruction is useful for the FORTRAN integer conversion standard, which requires rounding to zero.

Exceptions								**Data Formats**						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	0	0	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if an inexact conversion occurred. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Integer, rounded to zero		+ 0.0	−0.0	+infinity	− infinity

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	0	0	1	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FINTRZ FP0,FP1 instruction places the integer part of floating point data register 0 in floating point data register FP1, with forced rounding to zero. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	0	0	1	1

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1

The hex value is $F2000083. Executing the instruction yields:

Before:

```
USP=b03f8386 MSP=b03f8386 ISP =b03f8384 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8384
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 3fff c147ae147ae147ae Decimal: +1.5100000000000000E+000
FP1: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
PC =FINTRZ$     FINTRZ     FP0,FP1
```

After:

```
USP=b03f8386 MSP=b03f8386 ISP =b03f8384 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8384
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 3fff c147ae147ae147ae Decimal: +1.5100000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FINTRZ$+004 FNOP
```

Note that 1.51 rounded to 1.0, in spite of the fact that the Round to Nearest option was in effect.

FLOG10 Instruction

Logarithm, Base 10

Syntax:
FLOG10.f <ea>,FPn
FLOG10.X FPm,FPn
FLOG10.X FPn,FPn

Description

The FLOG10 instruction calculates the logarithm of the source operand using base 10 arithmetic. The function is not defined for source values less than zero.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	0	*	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source is less than zero. Cleared otherwise.
OVFL	Cleared.
UNFL	Cleared.
DZ	Set if source is +0 or -0. Cleared otherwise.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if an inexact conversion occurred. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	LOG_{10}	NAN(OPERR)	− infinity(DZ)		+ infinity	NAN(OPERR)

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An		xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	0	1	0	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FLOG10 FP0,FP1 instruction places the base 10 logarithm of the number in floating point data register 0 in floating point data register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	0	1	0	1

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	1

The hex value is $F2000095. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4005 c800000000000000 Decimal: +1.0000000000000000E+002
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FLOG10$     FLOG10     FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 4005 c800000000000000 Decimal: +1.0000000000000000E+002
FP1: 4000 8000000000000001 Decimal: +2.0000000000000000E+000
PC =FLOG10$+004 FNOP
```

This example calculates $\log_{10}$ of 100, yielding 2. Note that the inexact bit (INEX2) is set, even though the result is exact.

FLOG2 Instruction

Logarithm, Base 2

Syntax: **FLOG2.f <ea>,FPn**
FLOG2.X FPm,FPn
FLOG2.X FPn,FPn

Description

The FLOG2 instruction calculates the logarithm of the source operand using base 2 arithmetic. The function is not defined for source values less than zero.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	0	*	*	*	•	•	•	∘	∘	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source is less than zero. Cleared otherwise.
OVFL	Cleared.
UNFL	Cleared.
DZ	Set if source is +0 or -0. Cleared otherwise.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if an inexact conversion occurred. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	LOG 2	NAN(OPERR)	−infinity(DZ)		+ infinity	NAN(OPERR)

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	0	1	1	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	**Suffix**	**Data Type**
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point data register.

Example The FLOG2 FP0,FP1 instruction places the base 2 logarithm of the number in floating point data register FP0 in floating point data register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	0	1	1	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0

The hex value is $F2000096. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 400b 8000000000000000 Decimal: +4.0960000000000000E+003
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FLOG2$     FLOG2     FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 400b 8000000000000000 Decimal: +4.0960000000000000E+003
FP1: 4002 c000000000000000 Decimal: +1.2000000000000000E+001
PC =FLOG2$+004  FNOP
```

This example calculates $\log_2$ of 4096, yielding 12. Note that the inexact bit (INEX2) is set, even though the result is exact.

FLOGN Instruction

Logarithm, Base e

Syntax:
FLOGN.f <ea>,FPn
FLOGN.X FPm,FPn
FLOGN.X FPn,FPn

Description

The FLOGN instruction calculates the logarithm of the source operand using base e arithmetic. Such a logarithm is often termed a *natural logarithm.* The function is not defined for source values less than zero.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	0	*	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source is less than zero. Cleared otherwise.
OVFL	Cleared.
UNFL	Cleared.
DZ	Set if source is +0 or -0. Cleared otherwise.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if an inexact conversion occurred. Cleared otherwise.

Numerical Results

Destination	Source		
	ˋ In Range 1	ˋ Zero 1	ˋ Infinity 1
Result	LOG e NAN(OPERR)	1 infinity(DZ)	ˋ infinity NAN(OPERR)

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	0	1	0	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	**Suffix**	**Data Type**
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point data register.

Example The FLOGN FP0,FP1 instruction places the base e logarithm of the number in floating point data register 0 in floating point data register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	0	1	0	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0

The hex value is $F2000094. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 4006 9469c4cb819c78f7 Decimal: +1.4841315910257660E+002
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FLOGN$      FLOGN      FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 4006 9469c4cb819c78f7 Decimal: +1.4841315910257660E+002
FP1: 4001 a000000000000001 Decimal: +5.0000000000000000E+000
PC =FLOGN$+004  FNOP
```

The number in FP0 is e^5. The natural logarithm of this quantity is 5, the contents of FP1 following the instruction.

FLOGNP1 Instruction

Logarithm (x+1), Base e

Syntax: **FLOGNP1.f <ea>,FPn**
FLOGNP1.X FPm,FPn
FLOGNP1.X FPn,FPn

Description

The FLOGNP1 instruction calculates the logarithm of the source operand plus 1 using base e arithmetic. A base e logarithm is often termed a *natural logarithm.* The function is not defined for source values less than or equal to -1.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	0	*	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source is less than zero. Cleared otherwise.
OVFL	Cleared.
UNFL	Cleared.
DZ	Set if source is -1. Cleared otherwise.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if an inexact conversion occurred. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	LOG $_e$ (x+1)		+0.0	−0.0	+ infinity	NAN(OPERR)

Exception bits that are always set by a combination of operand ranges are indicated in parentheses. Values less than -1 set the OPERR bit. The result is a NAN in that case.

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An		xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	0	1	1	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	**Suffix**	**Data Type**
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FLOGNP1 FP0,FP1 instruction places the base e logarithm of the number in floating point data register FP0 plus 1 in floating point data register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	0	1	1	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0

The hex value is $F2000086. Executing the instruction yields:

Before:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000008 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 4006 9369c4cb819c78f7 Decimal: +1.4741315910257660E+002
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
PC =FLOGNP1$    FLOGNP1    FP0,FP1
```

After:

```
USP=b03f83a2 MSP=b03f83a2 ISP =b03f83a0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f83a0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 4006 9369c4cb819c78f7 Decimal: +1.4741315910257660E+002
FP1: 4001 a000000000000001 Decimal: +5.0000000000000000E+000
PC =FLOGNP1$+004 FNOP
```

The number in FP0 is e^5- 1. Log_e of this quantity plus 1 is 5, contained in FP1 following the instruction.

FMOD Instruction

Modulo Remainder

Syntax: **FMOD.f** **<ea>,FPn**
FMOD.X **FPm,FPn**

Description

The FMOD instruction calculates the modulus resulting from dividing the destination by the source operand. The source is converted to extended precision (if required) and the integer part of the remainder of the division (destination ÷ source) is placed in the destination register. The conversion to integer always uses the round to zero (chop) mode. A similar instruction, FREM, uses the round to nearest mode, as required by the IEEE standard. FMOD is not defined for a source operand equal to zero or for a destination operand equal to infinity.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source is zero or destination is infinity. Cleared otherwise.
OVFL	Cleared.
UNFL	Set if an underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if an inexact conversion occurred. Cleared otherwise.

The quotient byte (bits 23–16) of the Floating Point status register is set to the sign and least significant 7 bits of the division (destination ÷ source).

Numerical Results

Destination	Source		
	In Range + −	Zero + −	Infinity + −
In Range + −	Modulo Remainder	NAN (OPERR)	FPn
Zero + −	+0.0 − 0.0	NAN (OPERR)	−0.0 +0.0
Infinity + −	NAN (OPERR)	NAN (OPERR)	NAN(OPERR)

Exception bits that are always set by a combination of operand ranges are indicated in parentheses. When the source is infinity and the destination is in range, the original contents of the destination register are unchanged.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	0	0	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FMOD FP0,FP1 instruction computes the modulo remainder of the division (FP1 ÷ FP0). The result is placed in register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	0	0	1

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1

The hex value is $F20000A1. Executing the instruction yields:

Before:

```
USP=b03f832c MSP=b03f832c ISP =b03f832a SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f832a
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4001 a000000000000000 Decimal: +5.0000000000000000E+000
FP1: 4001 e000000000000000 Decimal: +7.0000000000000000E+000
PC =FMOD$      FMOD      FP0,FP1
```

After:

```
USP=b03f832c MSP=b03f832c ISP =b03f832a SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f832a
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00010000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4001 a000000000000000 Decimal: +5.0000000000000000E+000
FP1: 4000 8000000000000000 Decimal: +2.0000000000000000E+000
PC =FMOD$+004   FNOP
```

Note the 01 in the quotient byte of the FPSR. The remainder of the operation 7/5 is 2. It is in register FP1 after the instruction.

FMOVE Instruction (Control Reg)

Move Floating Point Control Register

Syntax: FMOVE.L <ea>,FPcr
FMOVE.L FPcr,<ea>

Description

The FMOVE instruction has three forms. The two described in the next section move and convert data between a floating point data register and an effective address. The form described here moves data between an effective address and a control register. There are three coprocessor control registers: the Floating Point Control Register (FPCR), Floating Point Status Register (FPSR), and Floating Point Instruction Address Register (FPIAR). These registers are described in the previous Register Set section.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
*	*	*	*	*	*	*	*			∘				

BSUN	Can be set by a move to FPSR. Unchanged otherwise.
SNAN	Can be set by a move to FPSR. Unchanged otherwise.
OPERR	Can be set by a move to FPSR. Unchanged otherwise.
OVFL	Can be set by a move to FPSR. Unchanged otherwise.
UNFL	Can be set by a move to FPSR. Unchanged otherwise.
DZ	Can be set by a move to FPSR. Unchanged otherwise.
INEX1	Can be set by a move to FPSR. Unchanged otherwise.
INEX2	Can be set by a move to FPSR. Unchanged otherwise.

The transfer size is restricted to a 32-bit longword. The FPCR contains only 16 bits. The missing 16 bits are ignored when the FPCR is written, and they read back as zeros. To ensure compatability with future devices, Motorola advises that you not write anything but zeros to these missing bits.

This instruction can cause no new exceptions to take place regardless of the values written to the FPCR Exception Enable byte or FPSR Exception Status Byte.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000	•	•
An	001	An	•	•	xxx.L	111	001	•	•
(An)	010	An	•	•	#<data>	111	100	•	
(An)+	011	An	•	•					
-(An)	100	An	•	•					
(d16,An)	101	An	•	•	(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•	•	(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•	•	(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•	•	([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•	•	([bd,PC],Xn,od)	111	011	•	

☐ Valid only for FPIAR control Register

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
1	Dir	0	Register Select			0	0	0	0	0	0	0	0	0	0

Bit 14 (Dir) of the extension word determines whether the operation is effective address to control register (Dir = 0) or control register to effective address (Dir = 1).

The Register Select field (bits 12–10 of the extension word) indicates the Control Register. The encoding for this field is

Value	**Mnemonic**	**Register Name**
100	FPCR	Floating Point Control Register
010	FPSR	Floating Point Status Register
001	FPIAR	Floating Point Instruction Address Register

Example The FMOVE #$FFFFFFFF,FPSR instruction sets all implemented bits in the Floating Point Status Register (in violation of the warning not to try to set unimplemented bits). This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
1	Dir	0	Register Select			0	0	0	0	0	0	0	0	0	0

1	1	1	1	0	0	1	0	0	0	1	1	1	1	0	0
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

The hex value is $F23C8800 FFFFFFFF. Executing the instruction yields:

Before:

```
USP=b03f8324 MSP=b03f8324 ISP =b03f8322 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8322
FPCR: 0000ff00 BSUN SNAN OPERR OVFL UNFL DZ INEX2 INEX1 Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
PC =FMOVE$CR    FMOVE       #$FFFFFFFF,FPSR
```

After:

```
USP=b03f8324 MSP=b03f8324 ISP =b03f8322 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8322
FPCR: 0000ff00 BSUN SNAN OPERR OVFL UNFL DZ INEX2 INEX1 Ext Round Nearest
FPSR: 0ffffff8 N Z I NAN ESTAT BSUN SNAN OPERR OVFL UNFL DZ INEX2 INEX1
      Accrued: IOP OVFL UNFL DZ INEX FPIAR: 00000000
PC =FMOVE$CR+008            FNOP
```

Note the unimplemented bits that were not set to 1 in the FPSR: the upper 4 bits and the lower 3 bits. Note also that you can set the exception status bits, but no new exception will occur as a result of this instruction, even though the exceptions are enabled (in FPCR).

FMOVE Instruction (Data Reg)

Move Floating Point Data Register

Syntax: FMOVE.f <ea>,FPn
FMOVE.f FPm,<ea>
FMOVE.P FPm,<ea>{Dn}
FMOVE.P FPm,<ea>{#k}

Description

The FMOVE instruction has three forms. The two described in this section move and convert data between a floating point data register and an effective address. The other form (described earlier) moves data between an effective address and a control register. When an operand is moved to a floating point data register, it is converted (if necessary) to extended format. The current rounding mode presides over this conversion. When moving from the register to memory, conversion is also performed. Moving from a register to a packed decimal memory operand allows format conversion specification useful in converting floating point data to a character representation.

<ea> to Register Form:

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	0	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Cleared.
UNFL	May be set if source is an extended precision denormalized number. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if Data format is P. Cleared otherwise.
INEX2	May be set if Data format is L, D, or X. Cleared otherwise.

You can get an inexact result on several combinations of source format and rounding mode (as contained in the FPCR mode control byte). These combinations are

	Source Format	**B**	**W**	**L**	**S**	**D**	**X**	**P**
Rounding	**Single**			⊚		⊚	⊚	⊚
Mode	**Double**						⊚	⊚
	Extended							⊚

All combinations that are not marked with a bullet (⊚) above produce an exact result.

Register to <ea> Form:

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	0	0	0	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source is infinity or if destination is too small to contain source. Cleared otherwise.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	Cleared.
INEX2	Set if an inexact conversion took place. Cleared otherwise.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000	•	•
An	001	An			xxx.L	111	001	•	•
(An)	010	An	•	•	#<data>	111	100	•	
(An)+	011	An	•	•					
-(An)	100	An	•	•					
(d16,An)	101	An	•	•	(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•	•	(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•	•	(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•	•	([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•	•	([bd,PC],Xn,od)	111	011	•	

☐ Valid only for byte, word, long, or single data formats

<ea> to Register Form:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td colspan="3">Mode</td><td colspan="3">Register</td></tr>
<tr><td>0</td><td>R/M</td><td>0</td><td colspan="3">Source Spec</td><td colspan="3">Dest Register</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FMOVE.L D0,FP0 instruction converts the contents of CPU register D0 to extended precision, leaving the result in floating point data register FP0. This instruction has the following layout:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td colspan="3">Mode</td><td colspan="3">Register</td></tr>
<tr><td>0</td><td>R/M</td><td>0</td><td colspan="3">Source Spec</td><td colspan="3">Dest Register</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The hex value is $F2004000. Executing the instruction yields:

Before:

```
USP=b03f832c MSP=b03f832c ISP =b03f832a SR =2700 -- -- SI PL=7 -----
D0: 0000000c 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f832a
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
PC =FMOVE$DRD   FMOVE.L    D0,FP0
```

After:

```
USP=b03f832c MSP=b03f832c ISP =b03f832a SR =2700 -- -- SI PL=7 -----
D0: 0000000c 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f832a
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4002 c000000000000000 Decimal: +1.2000000000000000E+001
PC =FMOVE$DRD+004           FNOP
```

This example moves the quantity 12 from D0.L to FP0.

Register to <ea> Form:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	1	1	Destination Fmt			Source Register			K-Factor (if Required)						

The Destination Format field (bits 12–10 of the extension word) indicates the operand format for the destination effective address. The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P{#k}	Packed Decimal Real with static K-factor
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111	.P{Dn}	Packed Decimal Real with dynamic K-factor

The Source Register (bits 9–7 of the extension word) contains the number of the source floating point register.

The K-Factor field is used only for packed decimal format destinations. For other data formats, this field should be set to zero. This quantity specifies either a static 7 bit K-factor or a data register number, depending on the Destination Format field described above. The dynamic form of the K-factor contains the K-factor in the low 7 bits of a data register. The data register number is contained in the upper three bits (6–4) of the K-Factor field in the extension word.

Both static and dynamic K-factors are 7-bit two's complement quantities, yielding a number between -64 and +63. This range is interpreted as follows:

Range	Meaning
-64 -> +00	Number of significant digits to the right of the decimal point (Fortran F format)
+01 -> +17	Number of significant digits in the mantissa (Fortran E format)
+18 -> +63	Sets OPERR bit in FPSR exception byte. Treated as +17.

Negative and zero values specify a number of digits in the mantissa, taking into account the exponent. For instance, the number 1234.5678 will have 4 significant digits (1.235E+3) due to the exponent if the K-factor is zero. A K-factor of -1 yields 5 significant digits (1.2346E+3), and so on. Remember that the numbers are rounded in this process.

Positive K-factors control the total number of digits in the mantissa. A K-factor of +1 in converting 1234.5678 yields a result of 1.0E+3 (one significant digit). A K-factor of +2 yields 1.2E+3, and so on.

Example The FMOVE FP0,D0 instruction converts the extended precision number in FP0 to a longword in CPU register D0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	1	1	Destination Fmt			Source Register			K–Factor (if Required)						

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0

The hex value is $F2006000. Executing the instruction yields:

Before:

```
USP=b03f832c MSP=b03f832c ISP =b03f832a SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f832a
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4000 a000000000000000  Decimal: +2.5000000000000000E+000
PC =FMOVE$DRS   FMOVE.L    FP0,D0
```

After:

```
USP=b03f832c MSP=b03f832c ISP =b03f832a SR =2700 -- -- SI PL=7 -----
D0: 00000002 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f832a
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 4000 a000000000000000  Decimal: +2.5000000000000000E+000
PC =FMOVE$DRS+004          FNOP
```

The quantity 2.5 is chopped to 2 in the process of moving. Note that the inexact condition is raised in the Floating Point Status Register (FPSR).

FMOVECR Instruction Syntax: FMOVECR.X #ccc,FPn

Move Constant to Floating Point Data Register

Description

The 68881/2 coprocessors have an onboard ROM that stores useful constants. The FMOVECR instruction puts one of these constants (selected by the immediate field) into a floating point data register. The constant is rounded to the precision specified in the FPCR mode control byte.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	0	0	0	0	0	0	*						•	

BSUN	Cleared.
SNAN	Cleared.
OPERR	Cleared.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	Cleared.
INEX2	Set if an inexact conversion occurred. Cleared otherwise.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	1	0	1	1	1	Dest Register			Rom Offset						

The Dest Register field (bits 9–7 of the extension word) indicates the destination Floating Point data register. The ROM Offset field indicates which constant to place in the data register. The allowed constants are as follows:

Offset	Constant	Hex Value	Decimal Value
$00	π	4000 c90fdaa22168c235	+3.1415926535897932E+000
$0B	$\mathrm{Log}_{10}(2)$	3ffd 9a209a84fbcff798	+3.0102999566398120E-001
$0C	e	4000 adf85458a2bb4a9a	+2.7182818284590452E+000
$0D	$\mathrm{Log}_2(e)$	3fff b8aa3b295c17f0bc	+1.4426950408889634E+000
$0E	$\mathrm{Log}_{10}(e)$	3ffd de5bd8a937287195	+4.3429448190325180E-001
$0F	0.0	0000 0000000000000000	+0.0000000000000000E+000
$30	ln(2)	3ffe b17217f7d1cf79ac	+6.9314718055994530E-001
$31	ln(10)	4000 935d8dddaaa8ac17	+2.3025850929940457E+000
$32	10^0	3fff 8000000000000000	+1.0000000000000000E+000
$33	10^1	4002 a000000000000000	+1.0000000000000000E+001
$34	10^2	4005 c800000000000000	+1.0000000000000000E+002
$35	10^4	400c 9c40000000000000	+1.0000000000000000E+004
$36	10^8	4019 bebc200000000000	+1.0000000000000000E+008
$37	10^{16}	4034 8e1bc9bf04000000	+1.0000000000000000E+016
$38	10^{32}	4069 9dc5ada82b70b59e	+1.0000000000000000E+032
$39	10^{64}	40d3 c2781f49ffcfa6d5	+1.0000000000000000E+064
$3A	10^{128}	41a8 93ba47c980e98ce0	+1.0000000000000000E+128
$3B	10^{256}	4351 aa7eebfb9df9de8e	+1.0000000000000000E+256
$3C	10^{512}	46a3 e319a0aea60e91c7	+1.0000000000000000E+512
$3D	10^{1024}	4d48 c976758681750c17	+1.0000000000000000E+1024
$3E	10^{2048}	5a92 9e8b3b5dc53d5de5	+1.0000000000000000E+2048
$3F	10^{4096}	7525 c46052028a20979b	+1.0000000000000000E+4096

Example The FMOVECR #0,FP0 instruction puts the value π in register FP0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	1	0	1	1	1	Dest Register			Rom Offset						

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0

The hex value is $F2005C00. Executing the instruction yields:

Before:

```
USP=b03f82ba MSP=b03f82ba ISP =b03f82b8 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f82b8
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
PC =FMOVECR$   FMOVECR    #$00,FP0  ; Pi
```

After:

```
USP=b03f82ba MSP=b03f82ba ISP =b03f82b8 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f82b8
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued: --- ---- ---- -- INEX FPIAR: 00000000
FP0: 4000 c90fdaa22168c235 Decimal: +3.1415926535897932E+000
PC =FMOVECR$+004          FNOP
```

FMOVEM Instruction (Control Reg) Syntax:FMOVEM.L <list>,<ea>

Move Multiple Floating Point Control Registers **FMOVEM.L <ea>,<list>**

Description

The FMOVEM instruction has two forms: one for floating point data registers (described later), and one for floating point control registers. The form described here moves one or more control registers to or from an effective address. There are three coprocessor control registers: the Floating Point Control Register (FPCR), Floating Point Status Register (FPSR), and Floating Point Instruction Address Register (FPIAR). These registers are described in the previous Register Set section.

The list operand consists of the control register mnemonics separated by slash (/) characters. For instance, to transfer all three registers, this field would be "FPCR/FPSR/FPIAR".

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
*	*	*	*	*	*	*	*			•				

BSUN	Can be set by a move to FPSR. Unchanged otherwise.
SNAN	Can be set by a move to FPSR. Unchanged otherwise.
OPERR	Can be set by a move to FPSR. Unchanged otherwise.
OVFL	Can be set by a move to FPSR. Unchanged otherwise.
UNFL	Can be set by a move to FPSR. Unchanged otherwise.
DZ	Can be set by a move to FPSR. Unchanged otherwise.
INEX1	Can be set by a move to FPSR. Unchanged otherwise.
INEX2	Can be set by a move to FPSR. Unchanged otherwise.

Registers are always read or written as 32-bit longwords. The FPCR contains only 16 bits. The missing 16 bits are ignored when the FPCR is written, and they read back as zeros. To ensure compatability with future devices, Motorola advises that you not write anything but zeros to these missing bits.

This instruction can cause no new exceptions to take place regardless of the values written to the FPCR Exception Enable Byte or FPSR Exception Status Byte.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn	•	•	xxx.W	111	000	•	•
An	001	An	•	•	xxx.L	111	001	•	•
(An)	010	An	•	•	#<data>	111	100	•	
(An)+	011	An	•	•					
-(An)	100	An	•	•					
(d16,An)	101	An	•	•	(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•	•	(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•	•	(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•	•	([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•	•	([bd,PC],Xn,od)	111	011	•	

☐ Valid only for single transfers

☐ Valid only for single transfer to FPIAR control Register

The registers are always stored in memory as FPCR, then FPSR, then FPIAR, in increasing address order. This is true regardless of the addressing mode used.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
1	0	Dir	Register List			0	0	0	0	0	0	0	0	0	0

Bit 13 (Dir) of the extension word determines whether the operation is effective address to register (Dir = 0) or control register to effective address (Dir = 1).

The Register List field (bits 12–10 of the extension word) indicates which Control Registers to transfer. The encoding for this field is .

Value	Mnemonic	Register Name
100	FPCR	Floating Point Control Register
010	FPSR	Floating Point Status Register
001	FPIAR	Floating Point Instruction Address Register

Note that this encoding is identical to the FMOVE instruction for control registers, and that a single register FMOVEM transfer is the same as the equivalent FMOVE instruction.

Example The FMOVEM.L (A0)+,FPCR/FPSR/FPIAR instruction loads all three control registers from the location pointed to by register A0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
1	0	Dir	Register List					0	0	0	0	0	0	0	0

1	1	1	1	0	0	1	0	0	0	0	1	1	0	0	0
1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0

The hex value is $F2189C00. Executing the instruction yields:

Before:

```
USP=b03f81b2 MSP=b03f81b2 ISP =b03f81b0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8e5c 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f81b0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
PC =FMOVEM$CR   FMOVEM     (A0)+,FPCR/FPSR/FPIAR
```

The memory area addressed by A0 has the following format:

```
b03f8e5c:     ffffffff ffffffff ffffffff          ................
```

After:

```
USP=b03f81b2 MSP=b03f81b2 ISP =b03f81b0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8e68 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f81b0
FPCR: 0000fff0 BSUN SNAN OPERR OVFL UNFL DZ INEX2 INEX1 Rsv Round +inf
FPSR: 0ffffff8 N Z I NAN ESTAT BSUN SNAN OPERR OVFL UNFL DZ INEX2 INEX1
      Accrued: IOP OVFL UNFL DZ INEX FPIAR: ffffffff
PC =FMOVEM$CR+004          FNOP
```

Note that the unimplemented bits of the registers read back as zero.

FMOVEM Instruction (Data Reg)

Move Multiple Floating Point Data Registers

Syntax:FMOVEM.X <list>,<ea>
FMOVEM.X Dn,<ea>
FMOVEM.X <ea>,<list>
FMOVEM.X <ea>,Dn

Description

The FMOVEM instruction has two forms: one for floating point data registers and one for floating point control registers (described earlier). The form described here moves one or more floating point data registers to or from an effective address.

The list operand consists of the data register mnemonics separated by slash (/) or minus (-) characters. A slash separates individual registers, while a dash indicates a range. For instance, to transfer all registers except FP3, the field would be "FP0-FP2/FP4-FP7".

The data register form uses the low 8 bits of the specified data register to form a mask. The static list uses 8 bits of the instruction for the mask. This mask has two different forms: one for predecrement and one for all other addressing modes. For the predecrement addressing mode, bit 0 corresponds to register FP7, and bit 7 to FP0. For all other addressing modes, this order is reversed (bit 0 corresponds to FP0, and bit 7 to FP7).

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
–	–	–	–	–	–	–	–						•	

BSUN	Unchanged.
SNAN	Unchanged.
OPERR	Unchanged.
OVFL	Unchanged.
UNFL	Unchanged.
DZ	Unchanged.
INEX1	Unchanged.
INEX2	Unchanged.

The transfer size is restricted to extended (96 bits in memory, 80 bits in the registers). No conversion is performed, and the data is simply moved.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.
Dn	000	Dn		
An	001	An		
(An)	010	An	•	•
(An)+	011	An	•	
-(An)	100	An		•
(d16,An)	101	An	•	•
(d8,An,Xn)	110	An	•	•
(bd,An,Xn)	110	An	•	•
([bd,An,Xn],od)	110	An	•	•
([bd,An],Xn,od)	110	An	•	•

Syntax	Mode	Reg	Source	Dest.
xxx.W	111	000	•	•
xxx.L	111	001	•	•
#<data>	111	100		
(d16,PC)	111	010	•	
(d8,PC,Xn)	111	011	•	
(bd,PC,Xn)	111	011	•	
([bd,PC,Xn],od)	111	011	•	
([bd,PC],Xn,od)	111	011	•	

The registers are always stored in memory FP0–FP7, in increasing address order. This is true regardless of the addressing mode used.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
1	1	Dir	Mode		0	0	0	Register List							

Bit 13 (Dir) of the extension word determines whether the operation is effective address to register (Dir = 0) or data register to effective address (Dir = 1).

The Mode field (bits 12–11 of the extension word) indicates the format of the register list. The encoding for this field is

Value	**Meaning**
00	Static register list, predecrement addressing mode
01	Dynamic register list, predecrement addressing mode
10	Static register list, all other addressing modes
11	Dynamic register list, all other addressing modes

The register list field has the following format:

7	6	5	4	3	2	1	0	
FP0	FP1	FP2	FP3	FP4	FP5	FP6	FP7	**Predecrement addressing mode**
FP7	FP6	FP5	FP4	FP3	FP2	FP1	FP0	**All other addressing modes**
0	Data Register			0	0	0	0	**Dynamic Register List (in Dn)**

A dynamic register list in the low byte of a data register has the same format as the first two static fields above.

Example The FMOVEM.L (A0),FP0-FP7 instruction loads all seven floating point data registers from the location pointed to by register A0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
1	1	Dir	Mode		0	0	0	Register List							

1	1	1	1	0	0	1	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	0	0	1	1	1	1	1	1	1	1

The hex value is $F210D0FF. Executing the instruction yields:

Before:

```
USP=b03f81b2 MSP=b03f81b2 ISP =b03f81b0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8eba 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f81b0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000 Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP2: 4000 8000000000000000 Decimal: +2.0000000000000000E+000
FP3: 4000 c000000000000000 Decimal: +3.0000000000000000E+000
FP4: 4001 8000000000000000 Decimal: +4.0000000000000000E+000
FP5: 4001 a000000000000000 Decimal: +5.0000000000000000E+000
FP6: 4001 c000000000000000 Decimal: +6.0000000000000000E+000
FP7: 4001 e000000000000000 Decimal: +7.0000000000000000E+000
PC =FMOVEM$DR   FMOVEM     (A0),FP0-FP7
```

The memory area addressed by A0 has the following format:

```
b03f8eba:      3fff0000 80000000 00000000 40020000 ?...........@...
b03f8eca:      a0000000 00000000 40030000 a0000000 ........@.......
b03f8eda:      00000000 40030000 f0000000 00000000 ....@...........
b03f8eea:      40040000 a0000000 00000000 40040000 @...........@...
b03f8efa:      c8000000 00000000 40040000 f0000000 ........@.......
b03f8f0a:      00000000 40050000 8c000000 00000000 ....@...........
b03f8f1a:      00000000 00000000 00000000 00000000 ................
b03f8f2a:      00000000 00000000 00000000 00000000 ................
b03f8f3a:      00000000 00000000 00000000 00000000 ................
```

After:

```
USP=b03f81b2 MSP=b03f81b2 ISP =b03f81b0 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8eba 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f81b0
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 3fff 8000000000000000 Decimal: +1.0000000000000000E+000
FP1: 4002 a000000000000000 Decimal: +1.0000000000000000E+001
FP2: 4003 a000000000000000 Decimal: +2.0000000000000000E+001
FP3: 4003 f000000000000000 Decimal: +3.0000000000000000E+001
FP4: 4004 a000000000000000 Decimal: +4.0000000000000000E+001
FP5: 4004 c800000000000000 Decimal: +5.0000000000000000E+001
FP6: 4004 f000000000000000 Decimal: +6.0000000000000000E+001
FP7: 4005 8c00000000000000 Decimal: +7.0000000000000000E+001
PC =FMOVEM$DR+004          FNOP
```

Note that the 16-bit fields in the memory form of the numbers are not present in the registers.

FMUL Instruction

Floating Point Multiply

Syntax: FMUL.f <ea>,FPn
FMUL.X FPm,FPn

Description

The FMUL instruction computes the product of the source and destination operands, leaving the result in the destination. The source is converted to extended precision (if required). This operation is not defined for zero times infinity.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	*	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set for 0 times infinity. Cleared otherwise.
OVFL	Set if an overflow occurred. Cleared otherwise.
UNFL	Set if an underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if an inexact conversion occurred. Cleared otherwise.

Numerical Results

Destination		Source					
		In Range		Zero		Infinity	
		+	−	+	−	+	−
In Range	+	Multiply		+0.0	−0.0	+infinity	− infinity
	−			−0.0	+0.0	−infinity	+ infinity
Zero	+	+ 0.0	−0.0	+ 0.0	−0.0	NAN(OPERR)	
	−	− 0.0	+0.0	−0.0	+0.0		
Infinity	+	+infinity	− infinity	NAN (OPERR)		+ infinity	−infinity
	−	−infinity	+ infinity			− infinity	+infinity

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	0	1	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	**Suffix**	**Data Type**
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FMUL FP0,FP1 instruction computes the product of FP0 and FP1. The result is placed in register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	0	1	1

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	1

The hex value is $F20000A3. Executing the instruction yields:

Before:

```
USP=b03f8168 MSP=b03f8168 ISP =b03f8166 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8166
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4000 8000000000000000 Decimal: +2.0000000000000000E+000
FP1: 4001 c000000000000000 Decimal: +6.0000000000000000E+000
PC =FMUL$       FMUL       FP0,FP1
```

After:

```
USP=b03f8168 MSP=b03f8168 ISP =b03f8166 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8166
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4000 8000000000000000 Decimal: +2.0000000000000000E+000
FP1: 4002 c000000000000000 Decimal: +1.2000000000000000E+001
PC =FMUL$+004   FNOP
```

This example multiplies 6 and 2 to get 12.

FNEG Instruction

Floating Point Negate

Syntax: **FNEG.f <ea>,FPn**
FNEG.X FPm,FPn
FNEG.X FPn

Description

The FNEG instruction places the negative of the source operand in the destination floating point data register. The source is converted to extended precision (if required). The single register form is derived by specifying the same register as both source and destination.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	0	0	0	*	0	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Cleared.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Negate		+0.0	−0.0	+infinity	− infinity

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	0	1	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	**Suffix**	**Data Type**
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FNEG FP0 instruction negates register FP0, leaving the result in register FP0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	1	0	1	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0

The hex value is $F200001A . Executing the instruction yields:

Before:

```
USP=b03f8168 MSP=b03f8168 ISP =b03f8166 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8166
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4000 8000000000000000 Decimal: +2.0000000000000000E+000
PC =FNEG$       FNEG       FP0
```

After:

```
USP=b03f8168 MSP=b03f8168 ISP =b03f8166 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8166
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 08000000 N - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: c000 8000000000000000 Decimal: -2.0000000000000000E+000
PC =FNEG$+004   FNOP
```

FNOP Instruction

Syntax: FNOP

Floating Point No Operation

Description

The FNOP instruction does not perform any real operation. However, this instruction is useful for synchronizing the main processor and the floating point unit. You are guaranteed that all previous instructions have been completed and all pending exceptions have been initiated at the beginning of the FNOP instruction.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

This is actually an FBF (non-signaling False) conditional branch instruction, with a displacement of zero.

Example The hex value of an FNOP instruction is $F2800000. Executing the instruction yields:

Before:

```
USP=b03f8168 MSP=b03f8168 ISP =b03f8166 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8166
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
PC =FNOP$       FNOP
```

After:

```
USP=b03f8168 MSP=b03f8168 ISP =b03f8166 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8166
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
PC =FNOP$+004   FNOP
```

Only the PC changes as a result of this instruction.

FREM Instruction
IEEE Remainder

Syntax: **FREM.f** **<ea>,FPn**
FREM.X **FPm,FPn**

Description

The FREM instruction calculates the modulus resulting from dividing the destination by the source operand. The source is converted to extended precision (if required), and the IEEE remainder of the division (destination ÷ source) is placed in the destination register. The division always uses the round to nearest mode, as required by the IEEE standard.

The IEEE remainder is defined as

$$FPn - (Source \times (INT(FPn \div Source))$$

where the division uses round to nearest mode. FREM is not defined for a source operand equal to zero or a destination operand equal to infinity.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source is zero or destination is infinity. Cleared otherwise.
OVFL	Cleared.
UNFL	Set if an underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if an inexact conversion occurred. Cleared otherwise.

The quotient byte (bits 23–16) of the Floating Point Status Register (FPSR) is set to the sign and least-significant 7 bits of the division (destination ÷ source).

Numerical Results

Destination		Source		
		In Range + −	Zero + −	Infinity + −
In Range	+ −	IEEI Remainder	NAN (OPERR)	FPn
Zero	+ −	+0.0 −0.0	NAN (OPERR)	−0.0 +0.0
Infinity	+ −	NAN (OPERR)	NAN (OPERR)	NAN(OPERR)

Exception bits that are always set by a combination of operand ranges are indicated in parentheses. When the source is infinity and the destination is in range, the original contents of the destination register are unchanged.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	1	0	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FREM FP0,FP1 instruction computes the IEEE remainder of the division (FP1 ÷ FP0). The result is placed in register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	1	0	1

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	1

The hex value is $F20000A5. Executing the instruction yields:

Before:

```
USP=b03f815a MSP=b03f815a ISP =b03f8158 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8158
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4000 a000000000000000 Decimal: +2.5000000000000000E+000
FP1: 4001 e000000000000000 Decimal: +7.0000000000000000E+000
PC =FREM$      FREM      FP0,FP1
```

After:

```
USP=b03f815a MSP=b03f815a ISP =b03f8158 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f8158
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 08030000 N - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued: --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4000 a000000000000000 Decimal: +2.5000000000000000E+000
FP1: bffe 8000000000000000 Decimal: -5.0000000000000000E-001
PC =FREM$+004  FNOP
```

Note the 03 in the quotient byte of the FPSR. The IEEE remainder of the operation 7÷2.5 is -0.5 and is in register FP1 after the instruction.

FRESTORE Instruction (Privileged) Syntax: FRESTORE <ea>

Restore Coprocessor State

Description

The FRESTORE instruction restores the internal state of the floating point coprocessor. This operation, together with the FSAVE and FMOVEM instructions, permits suspending one set of floating point operations, and starting (or continuing) another set. Such a saving and restoring process is called a *context switch* operation and is used by multitasking systems to give the appearance of running many programs simultaneously.

The FRESTORE instruction does not affect the programmer-visible portion of the floating point coprocessor, except for the NULL state. You usually use FRESTORE to restore information saved using the FSAVE instruction. The effective address operand must reside in main memory.

The normal restoration process is to load the data registers and control registers with two FMOVEM instructions and then load the internal state with an FRESTORE. Saving the state uses exactly the inverse: FSAVE, then control and data registers. You can load the control and data registers in any order since the FMOVEM instruction for data registers does not affect the control registers.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn	000	Dn	
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

Three formats, NULL, IDLE, and BUSY, can be restored. Diagrams of these formats appear as part of Figures 4-16 and 4-17. These frames have the following sizes (in bytes):

State Frame	68881	68882
NULL	4	4
IDLE	28 ($1C)	60 ($3C)
BUSY	184 ($B8)	216 ($D8)

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	1	Mode			Register		

Example The FRESTORE (A0) instruction restores the state from the memory area whose address is in register A0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	1	Mode			Register		
1	1	1	1	0	0	1	1	0	1	0	1	0	0	0	0

The hex value is $F350. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8b28 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 0000fff0 BSUN SNAN OPERR OVFL UNFL DZ INEX2 INEX1 Rsv Round +inf
FPSR: 0ffffff8 N Z I NAN ESTAT BSUN SNAN OPERR OVFL UNFL DZ INEX2 INEX1
      Accrued:  IOP OVFL UNFL DZ INEX FPIAR: fffffff
FP0: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000  Decimal: +1.0000000000000000E+000
FP2: 4000 8000000000000000  Decimal: +2.0000000000000000E+000
FP3: 4000 c000000000000000  Decimal: +3.0000000000000000E+000
FP4: 4001 8000000000000000  Decimal: +4.0000000000000000E+000
FP5: 4001 a000000000000000  Decimal: +5.0000000000000000E+000
FP6: 4001 c000000000000000  Decimal: +6.0000000000000000E+000
FP7: 4001 e000000000000000  Decimal: +7.0000000000000000E+000
PC =FRESTORE$   FRESTORE   (A0)
```

The memory area addressed by A0 contains the following data:

```
b03f8b28:        00000000
```

Note that this is a null state frame, which is four bytes long. Restoring this frame yields the following:

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8b28 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 7fff ffffffffffffffff  Decimal: +0.ffffffffffffffffE-fff
FP1: 7fff ffffffffffffffff  Decimal: +0.ffffffffffffffffE-fff
FP2: 7fff ffffffffffffffff  Decimal: +0.ffffffffffffffffE-fff
FP3: 7fff ffffffffffffffff  Decimal: +0.ffffffffffffffffE-fff
FP4: 7fff ffffffffffffffff  Decimal: +0.ffffffffffffffffE-fff
FP5: 7fff ffffffffffffffff  Decimal: +0.ffffffffffffffffE-fff
FP6: 7fff ffffffffffffffff  Decimal: +0.ffffffffffffffffE-fff
FP7: 7fff ffffffffffffffff  Decimal: +0.ffffffffffffffffE-fff
PC =FRESTORE$+002          FNOP
```

Note that the entire programmer-visible register set is affected. The control registers are all cleared, and the floating point data registers are all set to non-signaling NANs.

FSAVE Instruction (Privileged)

Syntax: FSAVE <ea>

Save Coprocessor State

Description

The FSAVE instruction saves the internal state of the floating point coprocessor. This operation, together with the FRESTORE and FMOVEM instructions, permits suspending one set of floating point operations and starting (or continuing) another set. Such a saving and restoring process is called a *context switch* operation and is used by multitasking systems to give the appearance of running many programs simultaneously. The effective address operand must reside in main memory.

The normal saving process is to save the internal state with an FSAVE instruction, then save the data registers and control registers with two FMOVEM instructions. Restoring the state uses exactly the inverse: Restore the control and data registers with a pair of FMOVEM instructions, then restore the internal state with FRESTORE. You can load the control and data registers in any order since the FMOVEM instruction for data registers does not affect the control registers.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn	000	Dn	
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	
(d16,PC)	111	010	
(d8,PC,Xn)	111	011	
(bd,PC,Xn)	111	011	
([bd,PC,Xn],od)	111	011	
([bd,PC],Xn,od)	111	011	

Three formats, NULL, IDLE, and BUSY, can result from an FSAVE instruction. Diagrams of these formats appear as part of Figure 4-16 and 4-17. These frames have the following sizes (in bytes):

State Frame	68881	68882
NULL	4	4
IDLE	28 ($1C)	60 ($3C)
BUSY	184 ($B8)	216 ($D8)

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	0	Mode			Register		

Example The FSAVE (A0) instruction saves the coprocessor state in the memory area whose address is in register A0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	0	Mode			Register		
1	1	1	1	0	0	1	1	0	0	0	1	0	0	0	0

The hex value is $F310. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8b3e 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000  Decimal: +1.0000000000000000E+000
FP2: 4000 8000000000000000  Decimal: +2.0000000000000000E+000
FP3: 4000 c000000000000000  Decimal: +3.0000000000000000E+000
FP4: 4001 8000000000000000  Decimal: +4.0000000000000000E+000
FP5: 4001 a000000000000000  Decimal: +5.0000000000000000E+000
FP6: 4001 c000000000000000  Decimal: +6.0000000000000000E+000
FP7: 4001 e000000000000000  Decimal: +7.0000000000000000E+000
PC =FSAVE$      FSAVE      (A0)
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: b03f8b3e 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000  Decimal: +1.0000000000000000E+000
FP2: 4000 8000000000000000  Decimal: +2.0000000000000000E+000
FP3: 4000 c000000000000000  Decimal: +3.0000000000000000E+000
FP4: 4001 8000000000000000  Decimal: +4.0000000000000000E+000
FP5: 4001 a000000000000000  Decimal: +5.0000000000000000E+000
FP6: 4001 c000000000000000  Decimal: +6.0000000000000000E+000
FP7: 4001 e000000000000000  Decimal: +7.0000000000000000E+000
PC =FSAVE$+002  FNOP
```

The memory area addressed by A0 contains the following data:

```
      b03f8b3e:     1f380000 88008800 0f800000 88000000  .8..............
      b03f8b4e:     00000000 b03fc202 00000000 00000000  .....?..........
      b03f8b5e:     38000000 40010000 7fff0000 0e0e0e0e  8...@... ........
      b03f8b6e:     00000000 00000000 5c0effff 00000000  ........\.......
```

Note that this is an IDLE frame from a 68882 coprocessor.

FSCALE Instruction

Scale Exponent

Syntax: FSCALE.f <ea>,FPn
FSCALE.X FPm,FPn

Description

The FSCALE instruction converts the source operand to an integer (if required) and adds that integer to the destination exponent. The integer conversion uses round to zero (chop) mode. If the absolute value of the source operand is greater than or equal to 2^{14}, underflow or overflow results. FSCALE provides a fast way to multiply or divide a floating point quantity by a power of two.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	*	*	0	*	0	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source is plus or minus infinity. Cleared otherwise.
OVFL	Set if an overflow occurred. Cleared otherwise.
UNFL	Set if an underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Cleared.

Numerical Results

Destination		Source		
		In Range + −	Zero + −	Infinity + −
In Range	+ −	Scale Exponent	FPn(INEX2 or UNFL)	NAN (OPERR)
Zero	+	+0.0	−0.0	NAN (OPERR)
	−	− 0.0	+0.0	
Infinity	+	+infinity	+infinity	NAN (OPERR)
	−	−infinity	−infinity	

Exception bits that are always set by a combination of operand ranges are indicated in parentheses. When the source is zero and the destination is in range, the original contents of the destination register are unchanged. If the rounding mode has changed since the register was loaded, however, an inexact result and/or underflow may result.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	1	1	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FSCALE FP0,FP1 instruction scales the exponent in FP1 by the integer value of FP0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	1	1	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	0

The hex value is $F20000A6. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued:  --- ---- ---- -- INEX FPIAR: 00000000
FP0: 4000 8000000000000000  Decimal: +2.0000000000000000E+000
FP1: 4000 c90fdaa22168c235  Decimal: +3.1415926535897932E+000
PC =FSCALE$     FSCALE     FP0,FP1
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000008 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- INEX FPIAR: 00000000
FP0: 4000 8000000000000000  Decimal: +2.0000000000000000E+000
FP1: 4002 c90fdaa22168c235  Decimal: +1.2566370614359173E+001
PC =FSCALE$+004            FNOP
```

This example multiplies the value π (in FP1 before the instruction) by 4.

FScc Instruction

Syntax: FScc <ea>

Description

The FScc instruction evaluates a floating point condition. The byte integer at the effective address is set to all ones ($FF) if the condition is true and zeros ($00) if the condition is false.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
•	–	–	–	–	–	–	–	•						

BSUN	Set if NAN condition code is set and condition selected is IEEE non-aware. Cleared otherwise.
SNAN	Not affected.
OPERR	Not affected.
OVFL	Not affected.
UNFL	Not affected.
DZ	Not affected.
INEX1	Not affected.
INEX2	Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	
(d16,PC)	111	010	
(d8,PC,Xn)	111	011	
(bd,PC,Xn)	111	011	
([bd,PC,Xn],od)	111	011	
([bd,PC],Xn,od)	111	011	

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	1	Mode			Register		
0	0	0	0	0	0	0	0	0	0	Conditional Predicate					

The Conditional Predicate contains the encoding for the coprocessor condition to be interrogated. These combinations are as follows:

Mnemonic	Mnemonic Expansion	Equation	Predicate
F	False	0	000000
EQ	Equal	Z	000001
OGT	Ordered Greater Than	~(NAN | Z | N)	000010
OGE	Ordered Greater Than or Equal	Z | ~(NAN | N)	000011
OLT	Ordered Less Than	N & ~(NAN | Z)	000100
OLE	Unordered or Less Than or Equal	Z | (N & ~NAN)	000101
OGL	Ordered Greater or Less Than	~(NAN | Z)	000110
OR	Ordered	~NAN	000111
UN	Unordered	NAN	001000
UEQ	Unordered or Equal	NAN | Z	001001
UGT	Unordered or Greater Than	NAN | ~(N | Z)	001010
UGE	Unordered or Greater Than or Equal	NAN | Z | N	001011
ULT	Unordered or Less Than	NAN | (N & ~Z)	001100
ULE	Unordered or Less Than or Equal	NAN | Z | N	001101
NE	Not Equal	~Z	001110
T	True	1	001111
SF	Signaling False	0	010000
SEQ	Signaling Equal	Z	010001
GT	Greater Than	~(NAN | Z | N)	010010
GE	Greater Than or Equal	Z | ~(NAN | N)	010011
LT	Less Than	N & ~(NAN | Z)	010100
LE	Less Than or Equal	Z | (N& ~NAN)	010101
GL	Greater or Less Than	~(NAN | Z)	010110
GLE	Greater, Less, or Equal	~NAN	010111
NGLE	Not (Greater, Less, or Equal)	NAN	011000
NGL	Not(Greater or Less Than)	NAN | Z	011001
NLE	Not (Less Than or Equal)	NAN | ~(N | Z)	011010
NLT	Not Less Than	NAN | (Z | ~N)	011011
NGE	Not (Greater Than or Equal)	NAN | (N & ~Z)	011100
NGT	Not Greater Than	NAN | Z | N	011101
SNE	Signaling Not Equal	~Z	011110
ST	Signaling True	1	011111

▭ Can set Status Register BSUN exception bit

The shaded conditions set the BSUN (Branch / Set on UNordered) bit in the FPSR if the NAN condition code is set. The conditions are explained in detail in the Architecture description earlier in this chapter.

Example The FSEQ D0 instruction sets the low byte of D0 if the Z condition code in the Floating Point Status Register is set. The low byte of D0 is cleared otherwise. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	1	Mode			Register		
0	0	0	0	0	0	0	0	0	0	Conditional Predicate					

1	1	1	1	0	0	1	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

The hex value is $F2400001. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
PC =FSCC$       FSEQ       D0
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000ff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
PC =FSCC$+004   FNOP
```

FSGLDIV Instruction

Syntax: FSGLDIV.f <ea>,FPn
FSGLDIV.X FPm,FPn

Floating Point Divide (Single Precision)

Description

The FSGLDIV instruction divides the destination floating point data register by the source operand. The result is rounded to single precision (regardless of the current rounding mode) and left in the destination floating point data register. Both the source and destination must be representable with a 24-bit mantissa. If this is not the case, the accuracy of the result is not guaranteed. This function is not defined for (0÷0) or (infinity÷infinity).

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	*	*	*	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set for 0 ÷ 0, or infinity ÷ infinity. Cleared otherwise.
OVFL	Set if an overflow occurred. Cleared otherwise.
UNFL	Set if an underflow occurred. Cleared otherwise.
DZ	Set if source operand is zero and destination is in range. Cleared otherwise.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination		Source					
		In Range		Zero		Infinity	
		+	−	+	−	+	−
In Range	+	Divide (Single Precision)		+infinity (DZ)	− infinity	+0.0	− 0.0
	−			−infinity	+ infinity	−0.0	+ 0.0
Zero	+	+0.0	−0.0	NAN (OPERR)		+0.0	−0.0
	−	−0.0	+0.0			−0.0	+0.0
Infinity	+	+infinity	−infinity	+infinity	−infinity	NAN (OPERR)	
	−	−infinity	+infinity	−infinity	+infinity		

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	1	0	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FSGLDIV FP0,FP1 instruction places the single precision quantity (FP1 ÷ FP0) in floating point data register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	1	0	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0

The hex value is $F20000A4. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4000 c000000000000000  Decimal: +3.0000000000000000E+000
FP1: 4000 8000000000000000  Decimal: +2.0000000000000000E+000
PC =FSGLDIV$    FSGLDIV    FP0,FP1
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued:  --- ---- ---- -- INEX FPIAR: 00000000
FP0: 4000 c000000000000000  Decimal: +3.0000000000000000E+000
FP1: 3ffe aaaaab0000000000  Decimal: +6.6666668653488160E-001
PC =FSGLDIV$+004           FNOP
```

This example divides 2 by 3, with the result in FP1 following the instruction. Note the accuracy of the repeating fraction in the result. This fraction is .6666666...., but in the single divide it is truncated to 24 bits (roughly 6–7 decimal digits).

FSGLMUL Instruction

Floating Point Multiply (Single Precision)

Syntax: FSGLMUL.f <ea>,FPn
FSGLMUL.X FPm,FPn

Description

The FSGLMUL instruction computes the product of the source and destination operands and leaves the result in the destination. The source is converted to extended precision before the operation (if required). The destination is rounded to single precision, regardless of the current rounding mode in the Floating Point Control Register (FPCR).

Both operands are assumed to be representable with 24 bits of mantissa. If this is not the case, the accuracy of the result cannot be guaranteed. The exponent can also exceed the representation allowed with single precision, regardless of the current rounding mode. This operation is not defined for zero times infinity.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set for 0 times infinity. Cleared otherwise.
OVFL	Cleared.
UNFL	Set if an underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if an inexact conversion occurred. Cleared otherwise.

Numerical Results

Destination		Source					
		In Range		Zero		Infinity	
		+	−	+	−	+	−
In Range	+	Multiply (Single Precision)		+ 0.0	−0.0	+infinity	−infinity
	−			− 0.0	+0.0	−infinity	+infinity
Zero	+	+0.0	−0.0	+ 0.0	−0.0	NAN (OPERR)	
	−	−0.0	+0.0	− 0.0	+0.0		
Infinity	+	+infinity	−infinity	NAN (OPERR)		+infinity	−infinity
	−	−infinity	+infinity			−infinity	+infinity

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An		xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	1	1	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is:

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FSGLMUL FP0,FP1 computes the single-precision product of FP0 and FP1. The result is placed in register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	0	1	1	1

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1

The hex value is $F20000A7. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4000 c000000000000000  Decimal: +3.0000000000000000E+000
FP1: 4000 8000000000000000  Decimal: +2.0000000000000000E+000
PC =FSGLMUL$    FSGLMUL    FP0,FP1
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4000 c000000000000000  Decimal: +3.0000000000000000E+000
FP1: 4001 c000000000000000  Decimal: +6.0000000000000000E+000
PC =FSGLMUL$+004           FNOP
```

This example multiplies 2 and 3 to get 6.

FSIN Instruction

Floating Sine Function

Syntax: **FSIN.f** **<ea>,FPn**
FSIN.X **FPm,FPn**
FSIN.X **FPn**

Description

The FSIN instruction takes the sine of a floating point number and places the result in a floating point data register. The source is converted to extended precision (if required) before use. This function is not defined if the source is +infinity or -infinity. If the source is not in the range [-2π to +2π], it is reduced to that range. Large arguments may lose accuracy in this reduction process, and arguments greater than approximately 10^{20} may lose all accuracy.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source operand is + infinity or -infinity. Cleared otherwise.
OVFL	Cleared.
UNFL	Set if source is +infinity or -infinity. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Sine		+0.0	−0.0	NAN(OPERR)	

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	1	1	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FSIN FP0,FP1 instruction places the sine of the value in floating point data register 0 in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	1	1	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	1	1	1	0

The hex value is $F200008E. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000  Decimal: +1.0000000000000000E+000
PC =FSIN$       FSIN       FP0,FP1
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
FP1: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
PC =FSIN$+004   FNOP
```

The sine of zero (in FP0) is 0.0, the value in FP1 following the instruction.

FSINCOS Instruction

Simultaneous Sine and Cosine Functions

Syntax: **FSINCOS.f** **<ea>,FPc:FPs**
FSINCOS.X **FPm,FPc:FPs**

Description

The FSINCOS instruction takes both the sine and cosine of a floating point number and places the results in two floating point data registers. Doing both functions in a single instruction results in a significant speed advantage over doing the functions as two separate instructions. The source is converted to extended precision (if required) before use. This function is not defined if the source is +infinity or -infinity. If the source is not in the range [-2π to $+2\pi$], it is reduced to that range. Large arguments may lose accuracy in this reduction process, and arguments greater than approximately 10^{20} may lose all accuracy.

Specifying the same register for both FPs (sine) and FPc (cosine) causes the register to have the sine value following the instruction. The cosine value is discarded in this case.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source operand is + infinity or -infinity. Cleared otherwise.
OVFL	Cleared.
UNFL	Set if source operand is +infinity or -infinity. Cleared otherwise
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
FPs	Sine		+0.0	+ 0.0	NAN (OPERR+UNFL)	
FPc	Cosine		+1.0	+1.0	NAN (OPERR+UNFL)	

Exception bits which are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	●	xxx.W	111	000	●
An	001	An		xxx.L	111	001	●
(An)	010	An	●	#<data>	111	100	●
(An)+	011	An	●				
-(An)	100	An	●				
(d16,An)	101	An	●	(d16,PC)	111	010	●
(d8,An,Xn)	110	An	●	(d8,PC,Xn)	111	011	●
(bd,An,Xn)	110	An	●	(bd,PC,Xn)	111	011	●
([bd,An,Xn],od)	110	An	●	([bd,PC,Xn],od)	111	011	●
([bd,An],Xn,od)	110	An	●	([bd,PC],Xn,od)	111	011	●

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
1	R/M	0	Source Spec			Register FPs			0	1	1	0	Register FPc		

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Register FPs field (bits 9–7 of the extension word) contains the number of the destination floating point register to receive the sine result. The Register FPc field (bits 2–0 of the extension word) specifies the destination register to receive the cosine result.

Example The FSINCOS FP0,FP1:FP2 instruction places the cosine of the value in floating point data register 0 in floating point data register 1 and the sine in FP2. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Register FPs			0	1	1	0	Register FPc		

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	1

The hex value is $F2000131. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000  Decimal: +1.0000000000000000E+000
FP2: 4000 8000000000000000  Decimal: +2.0000000000000000E+000
PC =FSINCOS$    FSINCOS    FP0,FP1:FP2
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000  Decimal: +1.0000000000000000E+000
FP2: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
PC =FSINCOS$+004           FNOP
```

The sine and cosine of zero are 0 (in FP2) and 1 (in FP1), respectively.

FSINH Instruction

Floating Hyperbolic Sine Function

Syntax: FSINH.f <ea>,FPn
FSINH.X FPm,FPn
FSINH.X FPn

Description

The FSINH instruction takes the hyperbolic sine of a floating point number and places the result in a floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	*	0	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Set if an overflow occurred. Cleared otherwise.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Hyperbolic Sine		+0.0	−0.0	+infinity	− infinity

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An		xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	0	0	1	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FSINH FP0,FP1 instruction places the hyperbolic sine of the value in floating point data register 0 in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	0	0	1	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0

The hex value is $F2000082. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000  Decimal: +1.0000000000000000E+000
PC =FSINH$     FSINH     FP0,FP1
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
FP1: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
PC =FSINH$+004  FNOP
```

The hyperbolic sine of zero (in FP0) is 0.0, the value in FP1 following the instruction.

FSQRT Instruction

Floating Square Root

Syntax:

FSQRT.f <ea>,FPn
FSQRT.X FPm,FPn
FSQRT.X FPn

Description

The FSQRT instruction places the square root of the source operand in the destination floating point data register. The source is converted to extended precision (if required) before use. The single register form is derived by specifying the same register as both source and destination. This function is not defined if the source operand is less than zero.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	0	0	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source operand is less than zero. Cleared otherwise.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Square Root	NAN(OPERR)	+0.0	NAN(OPERR)	+infinity	NAN (OPERR)

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	0	1	0	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	**Suffix**	**Data Type**
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FSQRT FP0,FP1 instruction takes the square root of the number in register FP0, leaving the result in register FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	0	1	0	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0

The hex value is $F2000084. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4001 8000000000000000  Decimal: +4.0000000000000000E+000
FP1: 3fff 8000000000000000  Decimal: +1.0000000000000000E+000
PC =FSQRT$     FSQRT     FP0,FP1
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4001 8000000000000000  Decimal: +4.0000000000000000E+000
FP1: 4000 8000000000000000  Decimal: +2.0000000000000000E+000
PC =FSQRT$+004  FNOP
```

FSUB Instruction

Floating Subtract

Syntax: FSUB.f <ea>,FPn
FSUB.X FPm,FPn

Description

The FSUB instruction subtracts an operand from a floating point data register. The source operand is converted to extended precision (if required) before use.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	*	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if adding +infinity to -infinity. Cleared otherwise.
OVFL	Set if overflow occurred. Cleared otherwise.
UNFL	Set if underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination		Source					
		In Range		Zero		Infinity	
		+	−	+	−	+	−
In Range	+	Subtract		Subtract		− infinity	+ infinity
	−						
Zero	+	Subtract		+0.0	− 0.0	− infinity	+ infinity
	−			− 0.0	+ 0.0		
Infinity	+	+ infinity		+ infinity		NAN (OPERR)	− infinity
	−	− infinity		− infinity		− infinity	NAN (OPERR)

Exception bits that are always set by a combination of operand ranges are indicated in parentheses. Subtracting +zero from +zero or -zero from -zero produces +zero in RN, RZ, and RP rounding modes, and -zero in RM mode.

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An	•	xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	1	0	0	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FSUB FP0,FP1 instruction subtracts FP0 from FP1, leaving the result in FP1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	0	1	0	0	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0

The hex value is $F20000A8. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4000 8000000000000000  Decimal: +2.0000000000000000E+000
FP1: 4001 8000000000000000  Decimal: +4.0000000000000000E+000
PC =FSUB$       FSUB       FP0,FP1
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4000 8000000000000000  Decimal: +2.0000000000000000E+000
FP1: 4000 8000000000000000  Decimal: +2.0000000000000000E+000
PC =FSUB$+004   FNOP
```

FTAN Instruction
Floating Tangent Function

Syntax: FTAN.f <ea>,FPn
FTAN.X FPm,FPn
FTAN.X FPn

Description

The FTAN instruction takes the tangent of a floating point number and places the result in a floating point data register. The source is converted to extended precision (if required) before use. This function is not defined if the source is +infinity or −infinity. If the source is not in the range [-π/2 to +π/2], it is reduced to that range. Large arguments may lose accuracy in this reduction process, and arguments greater than approximately 10^{20} may lose all accuracy.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	*	*	*	0	*	*	●	●	●	●	●	●	●

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Set if source operand is + infinity or -infinity. Cleared otherwise.
OVFL	Set if overflow occurs. Cleared otherwise.
UNFL	Set if underflow occurs. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	Tangent		+0.0	+ 0.0	NAN (OPERR)	

Exception bits that are always set by a combination of operand ranges are indicated in parentheses.

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	•	xxx.W	111	000	•
An	001	An		xxx.L	111	001	•
(An)	010	An	•	#<data>	111	100	•
(An)+	011	An	•				
-(An)	100	An	•				
(d16,An)	101	An	•	(d16,PC)	111	010	•
(d8,An,Xn)	110	An	•	(d8,PC,Xn)	111	011	•
(bd,An,Xn)	110	An	•	(bd,PC,Xn)	111	011	•
([bd,An,Xn],od)	110	An	•	([bd,PC,Xn],od)	111	011	•
([bd,An],Xn,od)	110	An	•	([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	1	1	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FTAN FP0,FP1 instruction places the tangent of the value in floating point data register 0 in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	1	1	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1	1	1	1

The hex value is $F200008F. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
FP1: 3fff 8000000000000000  Decimal: +1.0000000000000000E+000
PC =FTAN$       FTAN       FP0,FP1
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
FP1: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
PC =FTAN$+004   FNOP
```

The tangent of zero (in FP0) is 0.0, the value in FP1 following the instruction.

FTANH Instruction

Floating Hyperbolic Tangent Function

Syntax:
FTANH.f <ea>,FPn
FTANH.X FPm,FPn
FTANH.X FPn

Description

The FTANH instruction takes the hyperbolic tangent of a floating point number and places the result in a floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	0	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Cleared.
UNFL	Set if an underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	Hyperbolic Tangent		+0.0	− 0.0	+1.0	−1.0

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	0	0	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FTANH FP0,FP1 instruction places the hyperbolic tangent of the value in floating point data register 0 in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	0	1	0	0	1

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1

The hex value is $F2000089. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 3fff 8000000000000000  Decimal: +1.0000000000000000E+000
FP1: 3fff 8000000000000000  Decimal: +1.0000000000000000E+000
PC =FTANH$      FTANH      FP0,FP1
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued:  --- ---- ---- -- INEX FPIAR: 00000000
FP0: 3fff 8000000000000000  Decimal: +1.0000000000000000E+000
FP1: 3ffe c2f7d5a8a79ca2ad  Decimal: +7.6159415595576490E-001
PC =FTANH$+004  FNOP
```

This example takes the hyperbolic tangent of 1.0, leaving the result in FP1 following the instruction.

FTENTOX Instruction

10^x

Syntax:
FTENTOX.f <ea>,FPn
FTENTOX.X FPm,FPn
FTENTOX.X FPn

Description

The FTENTOX instruction converts the source operand to extended precision (if required) and calculates 10 to that power. The result is placed in the destination floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	*	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Set if an overflow occurred. Cleared otherwise.
UNFL	Set if an underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	10^x		+1.0		+ Infinity	+0.0

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	0	0	1	0

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FTENTOX FP0,FP1 instruction takes 10 to the power of the value in floating point data register 0. The result is placed in floating point data register 1. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	0	0	1	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0

The hex value is $F2000092. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 4001 8000000000000000  Decimal: +4.0000000000000000E+000
FP1: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
PC =FTENTOX$    FTENTOX    FP0,FP1
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued:  --- ---- ---- -- INEX FPIAR: 00000000
FP0: 4001 8000000000000000  Decimal: +4.0000000000000000E+000
FP1: 400c 9c3fffffffffffff  Decimal: +1.0000000000000000E+004
PC =FTENTOX$+004           FNOP
```

This example calculates 10^4, leaving the result (10,000) in FP1 following the instruction.

FTRAPcc Instruction

TRAP on Floating Point Condition

Syntax: **FTRAPcc**
FTRAPcc.W #<data>
FTRAPcc.L #<data>

Description

The FTRAPcc instruction causes an exception through vector 7 (offset $1C) if the specified floating point condition is true. The FTRAPcc exception uses a format 2 stack frame, shown in Figure 2-39. The Macintosh MPW system uses FTPcc as the instruction mnemonic rather than FTRAPcc.

FTRAPcc instructions can be followed by one or two words of immediate data. The CPU adjusts the return address in the stack frame to skip over this immediate data upon executing an RTE. Note that the last two words in the stack frame (offsets $8 and $A) contain the address of the instruction word to assist in retrieving this immediate data.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
*	–	–	–	–	–	–	–		•	•				

BSUN	Set if NAN condition code is set and condition selected is IEEE non-aware. Cleared otherwise.
SNAN	Not affected.
OPERR	Not affected.
OVFL	Not affected.
UNFL	Not affected.
DZ	Not affected.
INEX1	Not affected.
INEX2	Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	1	1	1	1	Mode		
0	0	0	0	0	0	0	0	0	0	Conditional Predicate					

The mode field (bits 2–0 of the instruction word) specifies the size of the immediate data that follows the instruction. This field is 010 for an immediate word, 011 for an immediate long, and 100 for no immediate data.

The Conditional Predicate contains the encoding for the coprocessor condition to be interrogated. These combinations are as follows:

Mnemonic	Mnemonic Expansion	Equation	Predicate
F	False	0	000000
EQ	Equal	Z	000001
OGT	Ordered Greater Than	~(NAN | Z | N)	000010
OGE	Ordered Greater Than or Equal	Z | ~(NAN | N)	000011
OLT	Ordered Less Than	N & ~(NAN | Z)	000100
OLE	Unordered or Less Than or Equal	Z | (N & ~NAN)	000101
OGL	Ordered Greater or Less Than	~(NAN | Z)	000110
OR	Ordered	~NAN	000111
UN	Unordered	NAN	001000
UEQ	Unordered or Equal	NAN | Z	001001
UGT	Unordered or Greater Than	NAN | ~(N | Z)	001010
UGE	Unordered or Greater Than or Equal	NAN | Z | N	001011
ULT	Unordered or Less Than	NAN | (N & ~Z)	001100
ULE	Unordered or Less Than or Equal	NAN | Z | N	001101
NE	Not Equal	~Z	001110
T	True	1	001111
SF	Signaling False	0	010000
SEQ	Signaling Equal	Z	010001
GT	Greater Than	~(NAN | Z | N)	010010
GE	Greater Than or Equal	Z | ~(NAN | N)	010011
LT	Less Than	N & ~(NAN | Z)	010100
LE	Less Than or Equal	Z | (N& ~NAN)	010101
GL	Greater or Less Than	~(NAN | Z)	010110
GLE	Greater, Less, or Equal	~NAN	010111
NGLE	Not (Greater, Less, or Equal)	NAN	011000
NGL	Not(Greater or Less Than)	NAN | Z	011001
NLE	Not (Less Than or Equal)	NAN | ~(N | Z)	011010
NLT	Not Less Than	NAN | (Z | ~N)	011011
NGE	Not (Greater Than or Equal)	NAN | (N & ~Z)	011100
NGT	Not Greater Than	NAN | Z | N	011101
SNE	Signaling Not Equal	~Z	011110
ST	Signaling True	1	011111

▭ Can set Status Register BSUN exception bit

The shaded conditions set the BSUN (Branch / Set on UNordered) bit in the FPSR if the NAN condition code is set. The conditions are explained in detail in the Architecture description earlier in this chapter.

Example The FTRAPEQ instruction causes an exception through vector number 7 (offset $1C) if the Z condition code bit is not set. This instruction has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	1	1	1	1	Mode		
0	0	0	0	0	0	0	0	0	0	Conditional Predicate					

1	1	1	1	0	0	1	0	0	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

The hex value of the instruction is $F27C0001. Execution yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
PC =FTRAPCC$    FTRAPEQ
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7cfa SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7cfa
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 04000000 - Z - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
PC =TRAP7$      NOP
```

The exception took place because the Z bit was set.

FTST Instruction

Floating Test

Syntax: **FTST.f** **<ea>**
FTST.X **FPm**

Description

The FTST instruction performs a floating point test that is similar to the TST instruction for integer data types. The source operand is converted to extended precision (if necessary), and the floating point condition codes are set as a result of this operand. The source operand can be either an effective address operand or a floating point data register. The operand is not modified.

The Macintosh MPW system uses FTEST as the instruction mnemonic rather than FTST.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	•	0	0	0	0	•	0	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Cleared.
UNFL	Cleared.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Cleared.

Condition Codes

Destination	Source					
	In Range +	In Range −	Zero +	Zero −	Infinity +	Infinity −
Result	none	N	Z	N Z	I	NI

The indicated condition codes are always set when the operand is in the range in question. The NAN bit is set if the source operand is a NAN. All other condition code bits are cleared.

Valid Addressing Modes

Syntax	Mode	Reg	Valid	Syntax	Mode	Reg	Valid
Dn*	000	Dn	●	xxx.W	111	000	●
An	001	An		xxx.L	111	001	●
(An)	010	An	●	#<data>	111	100	●
(An)+	011	An	●				
-(An)	100	An	●				
(d16,An)	101	An	●	(d16,PC)	111	010	●
(d8,An,Xn)	110	An	●	(d8,PC,Xn)	111	011	●
(bd,An,Xn)	110	An	●	(bd,PC,Xn)	111	011	●
([bd,An,Xn],od)	110	An	●	([bd,PC,Xn],od)	111	011	●
([bd,An],Xn,od)	110	An	●	([bd,PC],Xn,od)	111	011	●

* Valid only for byte, word, or long data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	1	1	0	1	0

Bit 14 (R/M) of the extension word determines whether the source operand is a floating point register (R/M = 0) or an effective address operand (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register form (R/M = 0) or the operand format for the memory form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) is unused for the FTST instruction. These bits should be set to zero to maintain compatibility with future devices.

Example The FTST FP0 instruction sets the floating point condition codes according to the value in floating point data register 0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	1	1	1	0	1	0

1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0

The hex value is $F200003A. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: bfff 8000000000000000  Decimal: -1.0000000000000000E+000
PC =FTST$       FTST       FP0
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 08000000 N - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: bfff 8000000000000000  Decimal: -1.0000000000000000E+000
PC =FTST$+004   FNOP
```

The N condition code in the Floating Point Status Register is set because FP0 is less than zero.

FTWOTOX Instruction

2^X

Syntax:

FTWOTOX.f	**<ea>,FPn**
FTWOTOX.X	**FPm,FPn**
FTWOTOX.X	**FPn**

Description

The FTWOTOX instruction converts the source operand to extended precision (if required) and calculates 2 to that power. The result is placed in the destination floating point data register.

Exceptions								Data Formats						
BSUN	SNAN	OPERR	OVFL	UNFL	DZ	INEX1	INEX2	Byte	Word	Long	Sgl	Dbl	Ext	Pack
0	*	0	*	*	0	*	*	•	•	•	•	•	•	•

BSUN	Cleared.
SNAN	Set if source operand is an SNAN. Cleared otherwise.
OPERR	Cleared.
OVFL	Set if an overflow occurred. Cleared otherwise.
UNFL	Set if an underflow occurred. Cleared otherwise.
DZ	Cleared.
INEX1	May be set if memory operand is a packed decimal number. Cleared otherwise.
INEX2	Set if result is inexact. Cleared otherwise.

Numerical Results

Destination	Source					
	In Range		Zero		Infinity	
	+	−	+	−	+	−
Result	2^x		+1.0		+infinity	+0.0

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn*	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

* Valid only for byte, word, long, or single data types

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	0	0	0	1

Bit 14 (R/M) of the extension word determines whether the operation is register to register (R/M = 0) or memory to register (R/M = 1). The effective address field (bits 5–0 of the first instruction word) is valid only if the R/M bit is a 1. For the register-to-register form (R/M = 0), the effective address field should be zeros.

The Source Spec field (bits 12–10 of the extension word) indicates the register number for the register-to-register form (R/M = 0) or the operand format for the memory-to-register form (R/M = 1). The encoding for this field is

Value	Suffix	Data Type
000	.L	Longword (32-bit) Integer
001	.S	Single Precision Real
010	.X	Extended Precision Real
011	.P	Packed Decimal Real
100	.W	Word (16-bit) Integer
101	.D	Double Precision Real
110	.B	Byte (8-bit) Integer
111		(Reserved)

The Dest Register (bits 9–7 of the extension word) contains the number of the destination floating point register.

Example The FTWOTOX.B #$0C,FP0 instruction raises 2 to the twelfth power. The result is placed in floating point data register 0. This instruction has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	0	Mode			Register		
0	R/M	0	Source Spec			Dest Register			0	0	1	0	0	0	1

1	1	1	1	0	0	1	0	0	0	1	1	1	1	0	0
0	1	0	1	1	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0

The hex value is $F23C 5811 000C. Executing the instruction yields:

Before:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000000 - - - --- ESTAT ---- ---- ----- ---- ---- -- ----- -----
      Accrued:  --- ---- ---- -- ---- FPIAR: 00000000
FP0: 0000 0000000000000000  Decimal: +0.0000000000000000E+000
PC =FTWOTOX$    FTWOTOX.B  #$0C,FP0
```

After:

```
USP=b03f7d08 MSP=b03f7d08 ISP =b03f7d06 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 b03f7d06
FPCR: 00000000 ---- ---- ----- ---- ---- -- ----- ----- Ext Round Nearest
FPSR: 00000208 - - - --- ESTAT ---- ---- ----- ---- ---- -- INEX2 -----
      Accrued:  --- ---- ---- -- INEX FPIAR: 00000000
FP0: 400b 8000000000000000  Decimal: +4.0960000000000000E+003
PC =FTWOTOX$+006           FNOP
```

This example calculates 2^{12}, leaving the result (4,096) in FP0 following the instruction. Note that the INEX2 bit is set, even though the exponent is an integer.

CHAPTER

5

Memory Management Support

"We could manage this matter to a T." — Sterne

Background

Multitasking operating systems require hardware that will protect programs from each other. In an unprotected environment, like that found on early versions of the Apple Macintosh, Commodore Amiga, and IBM PC (using DOS), a program can destroy itself, another program, or even crash the entire system. Perhaps the worst scenario is that an errant program can destroy code or data, with no visible indication at the time. This situation is a time bomb that will explode sometime later. The potential for disaster is enormous.

In addition to a protected environment, some multitasking operating systems also support virtual memory. Although memory has become less expensive, main memory (RAM) is still far more expensive than disk memory. Virtual memory attempts to minimize the amount of RAM required by leaving portions of a program on disk until they are actually referenced.

Microcomputer users are not the first to address these problems. Memory protection was implemented on mainframes in the 1960s, on minicomputers in the 1970s, and finally on microcomputers in the 1980s. Support for virtual memory followed a similar pattern.

What is a Memory Management Unit?

A memory management unit (MMU) is a device that presents the illusion that software has access to memory that is arranged differently than the memory in the real machine. The MMU is interposed between the CPU and memory, as shown in Figure 5-1.

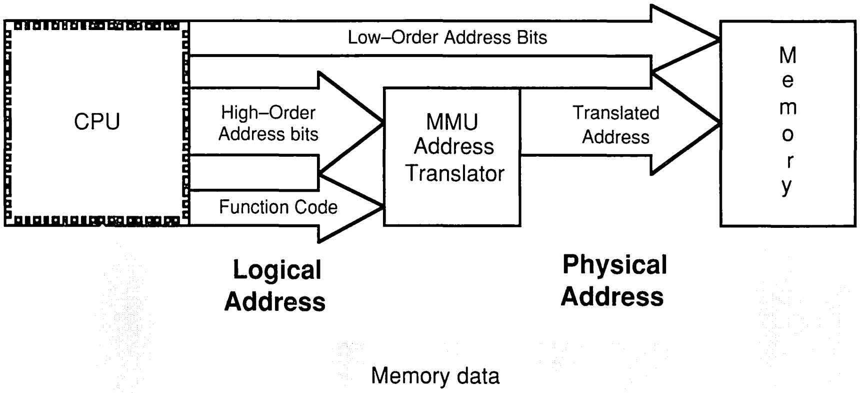

Low–Order Address Bits
CPU
High–Order Address bits
Function Code
MMU Address Translator
Translated Address
Memory
Logical Address
Physical Address
Memory data

Figure 5-1. MMU Overview

Some number of the low-order bits of the CPU-generated address (the *logical address*) are passed directly to the memory. The remaining address bits and the CPU function codes (FC2–FC0) cause the MMU to generate a translated address (a *physical address*). This physical address (which includes the low-order bits that do not participate in the translation) is presented to memory for the memory transfer operation.

Due to the presence of the function code bits, this structure makes it possible for supervisor mode addresses to translate differently than user mode addresses. Code and data references may also generate different translations. The number of address bits passed through without translation can be selected in software on Motorola MMUs. Other manufacturers impose a fixed number.

Memory Management Techniques

MMUs differ primarily in the technique used to translate logical addresses to physical addresses. The Motorola MMUs support two methods: *base/bounds* and *address translation cache*. These techniques are discussed below.

Base/Bounds Memory Management

Perhaps the simplest method of address translation is adding a number to the logical address to generate the corresponding physical address. The physical address is then compared to a limit to detect illegal memory references. This technique is called *base/bounds* memory management.

A base/bounds MMU is inexpensive because it requires only two registers:an adder, and a comparator. However, a single base/bounds area is not practical for most applications. Instead, it is usually desirable to split the memory required by a given program into multiple segments, as illustrated in Figure 5-2.

This figure represents the structure often used for high-level language programs in systems such as PC-DOS on the IBM PC and many versions of UNIX. The code area and data area may overlap if the MMU distinguishes between code and data references. Some systems (notably UNIX) use an area for data that has initial values and another area for data that has no initial values. The architecture of the MMU may require some vacant address space between the two.

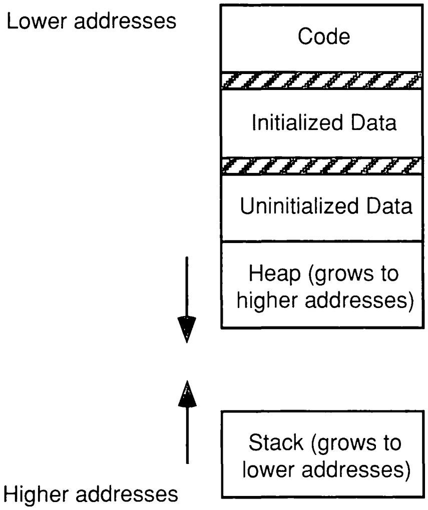

Lower addresses
Code
Initialized Data
Uninitialized Data
Heap (grows to higher addresses)
Stack (grows to lower addresses)
Higher addresses

Figure 5-2. Program Memory Areas

The end of the uninitialized data area can expand, usually toward higher addresses. This area is often called the *heap*. Memory allocation functions such as **new** in Pascal or **malloc** in C allocate memory in this area. The MMU must be able to change the size of this area when the program requests more heap memory.

The stack area (pointed to by register A7 on 68000 family chips) usually begins at the highest possible address and grows toward lower addresses. Many systems on 68000 family CPUs allocate a small area for the stack and enlarge this area when the program attempts to grow the stack outside this area. Thus, the MMU must also support the ability to expand an area from high addresses toward lower addresses.

The typical technique for implementing a base/bounds MMU is to take some number of high-order address bits, index into an array of base/bounds register pairs, and perform the translation as described above. Each of these base/bounds pairs corresponds to an area in physical memory called a *segment*. This layout permits placing each of the program areas described above in a separate segment. The segments can be protected and can change size independently. This architecture is shown in Figure 5-3.

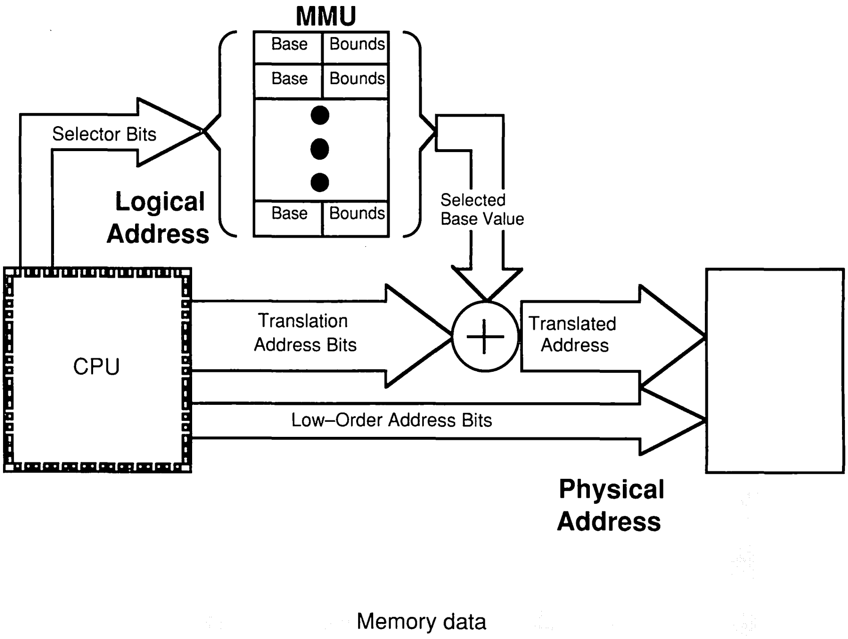

MMU
Base
Bounds
Base
Bounds
Base
Bounds
Selector Bits
Logical Address
Selected Base Value
CPU
Translation Address Bits
Translated Address
Low–Order Address Bits
Physical Address
Memory data

Figure 5-3. Segmented Base/Bounds MMU

Another advantage of the base/bounds technique is that the overhead introduced by the MMU is constant. The address translation cache technique described below does not offer this property. Performance using base/bounds is deterministic, but the address translation cache is statistical.

The major problem with a base/bounds architecture is the number of segments. One of the main goals of such an architecture is reducing the number of segments implemented in hardware. Regardless of the number chosen, it is never enough. The granularity may also present a problem. If you need a bit more than one segment permits, you must use two segments and further exacerbate the problem of the insufficient quantity of segments.

Address Translation Cache

From a programming point of view, the ideal situation is having a very large number of segments. With an extremely large number of segments, a bounds register per segment is unnecessary because all segments can be the same size. Most commercially available MMUs of this type center around fixed length segments (*pages*), which vary from 256 to 32768 (32K) bytes in size. Simple arithmetic indicates that you cannot implement such a system with a complete set of registers. (For instance, a 68020 with 32K pages requires 131,072 (128K) registers to map the 4Gb address space.) Since most paged systems use page sizes in the 2K to 8K range, it is too costly to implement a paged MMU completely with registers.

The approach taken by Motorola has similarities to the cache system used for main memory. The translation table is kept in main memory, with the most recently used entries maintained in hardware registers. The MMU then has the architecture shown in Figure 5-4.

The key feature of this architecture is the *address translation cache* (ATC). (This feature is called the *translation lookaside buffer* (TLB) in other architectures.) The ATC is a set of registers similar to the base/bounds pairs in Figure 5-3, with one notable exception. This cache is an associative memory since there is no simple way to tell whether the desired translation is in the cache. The logical address is presented to each cache entry in parallel. If the entry is in cache, the hardware presents the translated address as the physical address and the memory cycle completes normally.

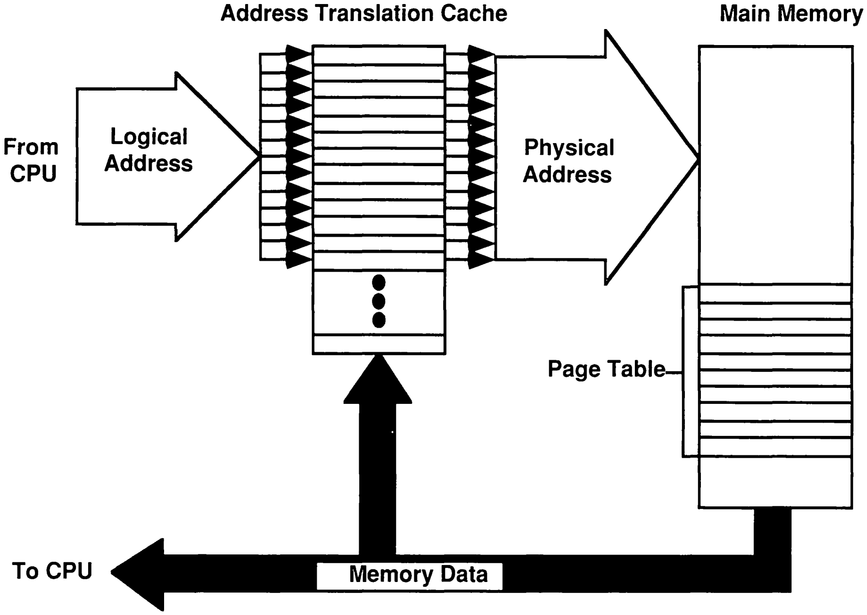

Address Translation Cache
Main Memory
From CPU
Logical Address
Physical Address
Page Table
To CPU
Memory Data

Figure 5-4 Address Translation Cache.

If the entry is not in cache, the MMU fetches the required entry from a main memory area called the *page table*, places this entry in cache, and completes the memory cycle as above. A program that addresses memory in an extremely random fashion could conceivably cause the MMU to access the page table on every memory reference, thereby cutting performance by at least a factor of two. Fortunately, this occurs infrequently and a cache hit rate of 95 per cent or better is typical.

The page table in this arrangement can be divided into segments just as in the base/bounds design so that missing portions of a program's address space are not represented in the page table. This subdivision of the address space can get quite complicated, as we shall soon see.

Motorola Memory Management Unit Architecture

Motorola has three memory management products for the 68000 family: the 68451, the 68851, and the 68851 subset, which is built into the 68030 processor chip.

68451

The 68451 was introduced at the same time as the 68010. This memory management unit was a base/bounds type with variable sized segments. Since it never gained widespread commercial acceptance, this book does not cover it.

68851

The 68851 MMU is the first Motorola device capable of supporting sophisticated demand paged virtual memory. This chip contains features previously found only in large-scale mainframe and minicomputer MMUs. The 68851 supports up to five levels of page table: one for the access type (supervisor/user, code/data, etc.) and four based on the address itself. You can specify the number of address bits used to index each table. There are sixty-four entries in the Address Translation Cache. This MMU offers a wide range of features in mapping a program's logical memory to physical memory.

The 68851 is designed for use with modern virtual memory software. This chip supports the module level protection mechanism associated with the 68020 CALLM/RETM instructions. External data cache mechanisms are provided to allow augmenting the 68020 with additional code and data cache support. Breakpoint support allows skipping a breakpoint for a preset number of counts, at very near full speed.

A rather distinctive (for microcomputers) feature in the 68851 is the ability to translate addresses provided by external devices. This feature allows an operating system to perform I/O to memory areas that are not contiguous in physical memory by giving the external device a logical rather than a physical address. This feature is known as *scatter/gather I/O* in mainframe and minicomputer systems. Since memory allocation on a virtual memory system is done using non-contiguous pages, this type of I/O is extremely beneficial to system designers.

68030

The 68030 MMU is a subset of the 68851. The desire to place both the processor and the CPU on a single chip forced severe restrictions on the functionality available for this MMU. The breakpoint feature, CALLM/RETM support, and DMA translation capability are not available on the 68030. Additionally, there are only twenty-two Address Translation Cache entries, as opposed to the sixty-four in the 68851 chip. The 68851 and 68030 share a common architecture, although the 68030 implements a subset of the 68851 has a *Transparent Translation* feature not found on the 68851.

Translation Table Overview

The key issue in using either MMU is constructing the translation table. You can have from one to five levels, with a software controlled allocation of bits in the logical address to page levels. Page sizes range from 256–32768 (0.25K–32K) bytes.

A register called a *root pointer* contains the physical address of the first level of the translation table. Each entry in this first level contains the address of a second-level table, each entry in the second level table points to a third level table, and so on. This structure appears in Figure 5-5.

The table can be up to five levels deep. Since not all entries need be present, the logical address range can have gaps. The level 0 table can be based either on the function codes FC2–FC0 or some number of address bits, as explained later.

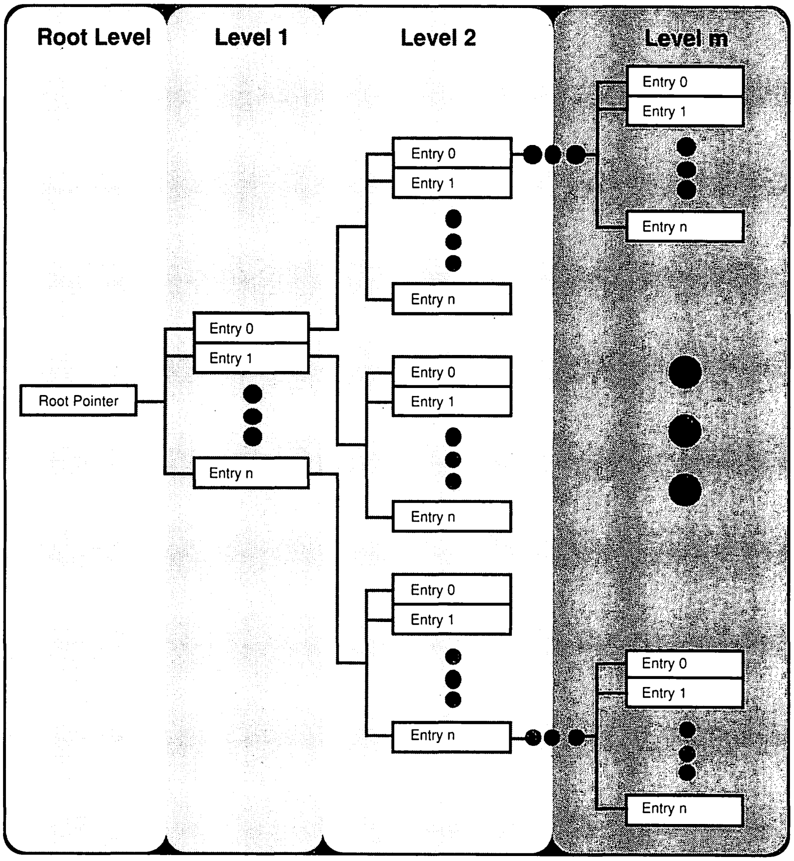

Root Level
Level 1
Level 2
Level m
Root Pointer
Entry 0
Entry 1
Entry n

Figure 5-5. Translation Table Overview

Register Set

The 68851 and 68030 share a common register architecture, with some registers on the 68851 not implemented on the 68030. The programmer-visible register set for both chips is shown in Figure 5-6.

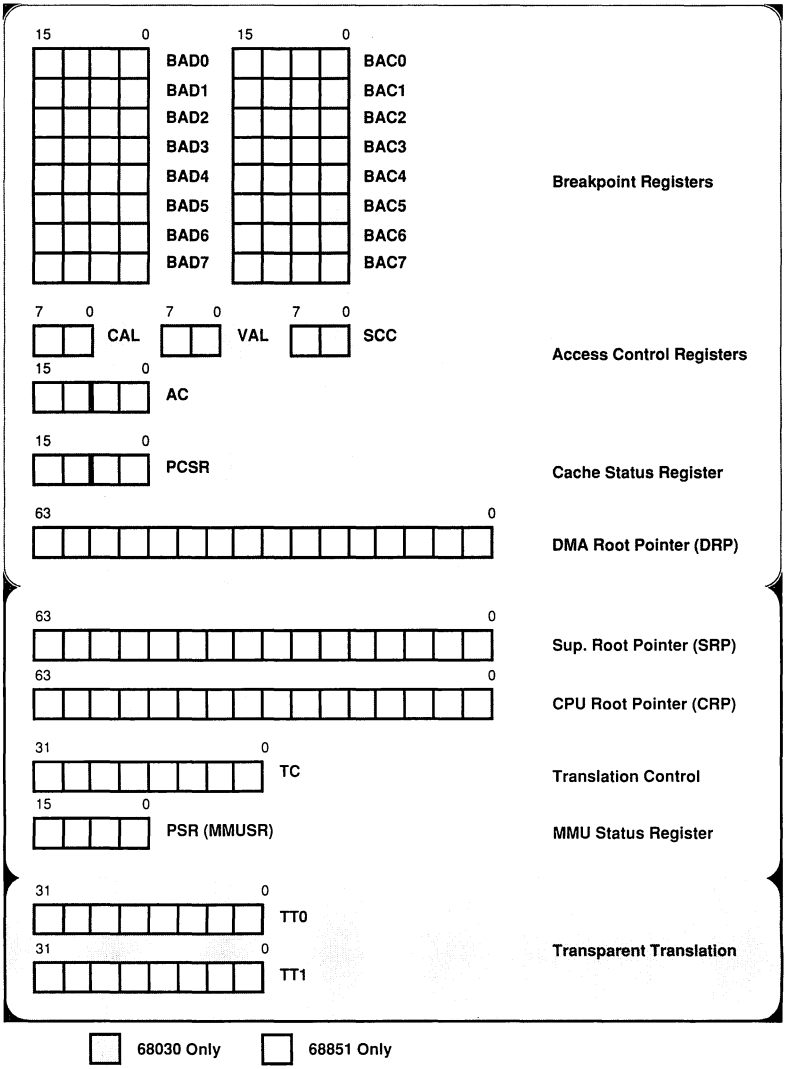

15
0
BAD0
BAD1
BAD2
BAD3
BAD4
BAD5
BAD6
BAD7
15
0
BAC0
BAC1
BAC2
BAC3
BAC4
BAC5
BAC6
BAC7
Breakpoint Registers
7
0
CAL
7
0
VAL
7
0
SCC
15
0
AC
Access Control Registers
15
0
PCSR
Cache Status Register
63
0
DMA Root Pointer (DRP)
63
0
Sup. Root Pointer (SRP)
63
0
CPU Root Pointer (CRP)
31
0
TC
Translation Control
15
0
PSR (MMUSR)
MMU Status Register
31
0
TT0
31
0
TT1
Transparent Translation
68030 Only
68851 Only

Figure 5-6. 68851/68030 Register Set

The 68851 has a much larger register set than the 68030 due to the fact that the 68851 is a separate chip. The breakpoint registers, access control registers, and DMA root pointer are all unavailable on the 68030. The transparent translation mechanism is unavailable on the 68851. The unshaded portion of the diagram represents registers available on both chips. Each is explained below.

MMU Status Register (PSR or MMUSR) The MMU status register is set by the PTEST instruction, which allows system software to validate an address supplied by a user program. This register is shown in Figure 5-7. This status register was called the Paged Memory Management Unit Register (PSR) in the 68851 documentation and the Memory Management Unit Status Register (MMUSR) in the 68030 documentation. This book uses the 68030 term.

Note that all bits are not implemented on both MMUs. Some bits are present only on the 68851, while others are present only on the 68030. Unimplemented bits read as zero and are ignored on write. You can specify the number of levels of page table that PTEST will access. Specifying zero levels means testing only the descriptors currently in the Address Translation Cache (ATC). The PTEST instruction also specifies the function code bits (FC2–FC0) and whether the access is to be a read or a write.

Bit 15 (**B**) indicates that the address being tested caused a BUSERR while fetching a descriptor. If only the ATC was tested, this bit indicates that there is no entry for the desired address in the ATC.

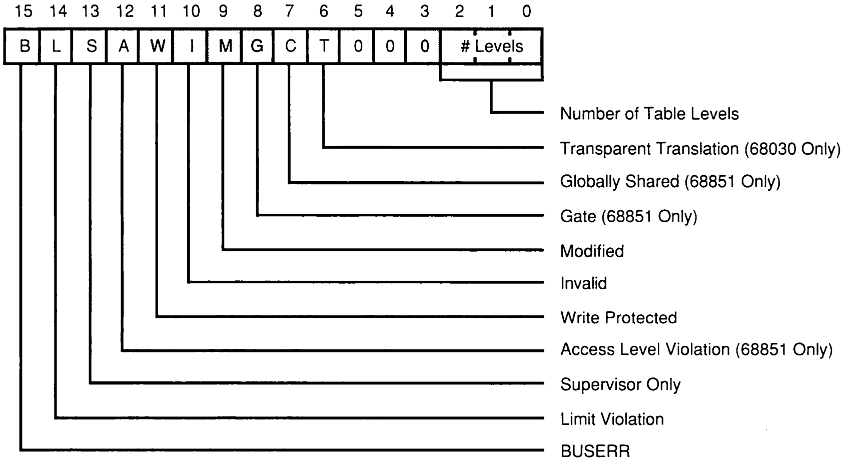

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
B
L
S
A
W
I
M
G
C
T
0
0
0
# Levels
Number of Table Levels
Transparent Translation (68030 Only)
Globally Shared (68851 Only)
Gate (68851 Only)
Modified
Invalid
Write Protected
Access Level Violation (68851 Only)
Supervisor Only
Limit Violation
BUSERR

Figure 5-7. MMUSR (PSR) Layout

Bit 14 (**L**) indicates that the PTEST operation found a limit violation on a long format descriptor.

Bit 13 (**S**) indicates that a Supervisor-only condition was violated. The descriptor in this case requires Supervisor mode, and the function code parameter to PTEST had bit 2 (FC2) equal to zero, indicating access from user mode.

Bit 12 (**A**) indicates a violation of the 68020 access level feature. Note that this bit is only supported for the 68851. The access violation is caused by a level mismatch based on the type of access requested (read or write).

Bit 11 (**W**) indicates a PTEST request for a write access on a descriptor that does not allow write accesses.

Bit 10 (**I**) indicates that the address requested cannot be translated legally.

Bit 9 (**M**) indicates that the translated address resides in a page that has been modified since the last time the modified bit in the descriptor was cleared.

Bit 8 (**G**) indicates that the translated address resides in a page that can contain CALLM module descriptors used for access level translations. These module descriptors are also called "gates," and are similar to the privilege level transition mechanisms used by other manufacturers. This bit is available only on the 68851 chip.

Bit 7 (**C**) indicates that the translated address resides in a page marked as globally shared. Note that this bit is only implemented for the 68851.

Bit 6 (**T**) indicates that the address is translated by one of the 68030 Transparent Translation registers.

Bits 2–0 (**# Levels**) indicate how many levels of translation table were searched. A value of zero means that the search terminated early or that the PTEST instruction specified a depth of zero (search ATC only).

Translation Control (TC) The Translation Control register determines the number of levels in the translation table and the number of bits to be used to index each level of the table. Figure 5-8 contains the layout of this register.

Bit 31 (**E**) is the enable bit. When this bit is written as a one, the MMU attempts to begin translation. Enable is cleared by a hardware reset and may be cleared by software to stop address translation. On the 68851, the Enable bit must be cleared before

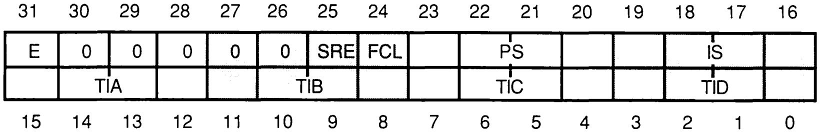


Figure 5-8. Translation Control (TC) Register

the contents of the Translation Control register can be updated. This restriction does not apply to the 68030. The 68030 Transparent Translation mechanism is not affected by the setting of this bit.

Bit 25 (**SRE**) is the Supervisor Root Enable bit. Setting this bit to one causes Supervisor mode accesses to be translated using the Supervisor Root Pointer (SRP) register. A zero value in this bit means that both supervisor and user accesses utilize the CPU Root Pointer (CRP) register. Note that a zero value for SRE means that both user and system modes share a common logical address space, unless the function code lookup feature is enabled (see discussion on the FCL bit below). Parts of a shared address space can be hidden from the user program using the supervisor-only feature of the translation descriptors.

Bit 24 (**FCL**) is the Function Code Lookup enable bit. If this bit is set to one, the first level translation table is indexed by the FC2–FC0 bits associated with the address to be translated. This feature allows you to separate code and data spaces. For the 68851 the additional FC3 bit distinguishes between main CPU and external device accesses. Setting the FCL bit causes the L/U and Limit fields of the root pointer registers (described below) to be ignored. When this bit is set, the first level page table has a maximum of eight entries for the CRP and SRP tables and eight entries for the DRP register on the 68851. The CRP and SRP use a table with the following organization:

Entry	FC3-0	Address Space	Root Pointer
0	0000	(Invalid, reserved for Motorola definition)	CRP
1	0001	User Data Space	CRP
2	0010	User Program (instruction) space	CRP
3	0011	(Invalid, Reserved for User definition)	CRP
4	0100	(Invalid, reserved for Motorola definition)	SRP or CRP
5	0101	Supervisor Data Space	SRP or CRP
6	0110	Supervisor Program (Instruction) space	SRP or CRP
7	0111	CPU Space	SRP or CRP

The choice between the Supervisor Root Pointer (SRP) and CPU Root Pointer (CRP) is made by the setting of the Supervisor Root Enable bit (SRE) of the Translation Control (TC) register. Normally, you would not use both the SRE and FCL bits simultaneously. Since it is usually sufficient to separate supervisor and user space with the separate translation tables allowed with the FCL bit, the SRP register is not normally used with the FCL feature enabled.

The DMA Root pointer is selected when function code bit FC3 has the value one. This table has the following organization:

Entry	FC3-0	Address Space	Root Pointer
0	1000	Alternate Bus master, user defined	DRP (68851 Only)
1	1001	Alternate Bus master, user defined	DRP (68851 Only)
2	1010	Alternate Bus master, user defined	DRP (68851 Only)
3	1011	Alternate Bus master, user defined	DRP (68851 Only)
4	1100	Alternate Bus master, user defined	DRP (68851 Only)
5	1101	Alternate Bus master, user defined	DRP (68851 Only)
6	1110	Alternate Bus master, user defined	DRP (68851 Only)
7	1111	Alternate Bus master, user defined	DRP (68851 Only)

Note that the entry numbers correspond only to the low three function code bits. There are a maximum of eight unique mappings at any given time using the DMA Root Pointer technique on the 68851. Thus, only 8 simultaneous transfers can take place into different address spaces with this technique. (For today's 68020 computers this is probably adequate.)

Bits 23–20 (**PS**) select the desired Page Size. This field contains the number of low-order address bits that are passed through the MMU without translation. Valid values of this field are

Value	Address Bits	Page Size
1000	8	256
1001	9	512
1010	10	1024
1011	11	2048
1100	12	4096
1101	13	8192
1110	14	16384
1111	15	32768

Note that bit 23 must always be a one. Writing a value of zero to the PS field with the Enable bit set will cause an MMU Configuration Exception through vector 56.

Bits 19–16 (**IS**) are the Initial Shift count. This field is analogous to the Page Size field, except that the bits that are not used by the MMU occur on the most significant side of the address and are discarded. This feature is useful in limiting the size of the translation table where the amount of logical address space required is less than 32 bits.

This feature is also required for running software developed on the 68000 and 68010 that uses the upper byte of addresses for something other than address bits. Macintosh software is a prime offender, using the upper 8 bits of an address as flag bits for the data indicated by the address. As a result, the "24-bit" and "32-bit" software modes on the Macintosh optionally discard the upper 8 address bits.

Bits 15–12 (**TIA**), bits 11–8 (**TIB**), bits 7–4 (**TIC**), and bits 3–0 (**TID**) specify the number of bits at each level of the translation table. Level A uses the highest order address bits, level B the next, and so forth. If a level is specified to use zero bits, that level and all below it are ignored. The sum of the PS, IS, TIA, TIB, TIC, and TIC fields is required to equal 32. A configuration exception results if this is not the case.

Translation Table Root Pointers (CRP, SRP, and DRP) There are two root pointers on the 68030 and three on the 68851. All have the format shown in Figure 5-9.

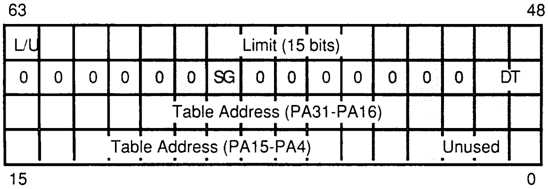


Figure 5-9. Root Pointer Register Layout

The address of the first level table is contained in the third and fourth words of this structure, in bits 31–4. The low 4 bits of the address are unused and must be zero. Note that this forces tables to be aligned on 16-byte boundaries.

Bit 63 (**L/U**) is the Lower/Upper bit. This bit controls the interpretation of the limit field. If this bit is a one, then the root pointer contains the address of a grow-down table. All entries using this root pointer must have values greater than or equal to the limit field. When the L/U bit is a one, all entries must have values less than or equal to the limit value. This feature is not normally used for a root pointer register, but has great value when applied to a translation table for a stack area, as described below.

Bits 62–48 (**Limit**) are a 15-bit unsigned binary quantity containing the lowest or highest legitimate value for the table addressed by the root pointer register, as described previously.

Bit 41 (**SG**), when set, indicates that the translation table data is shared globally by all tasks and that the MMU need only create one set of descriptors in the Address Translation Cache (ATC) for entries from this table. This bit is implemented only on the 68851 and is zero on the 68030.

Bits 33–32 (**DT**) determine the Descriptor Type of the register itself or of the table whose address is in bits 31–4. For root pointer registers, the following descriptor type values are valid:

DT Field	Meaning
$0	This value is used for invalid descriptors and may not be used for a root pointer register.
$1	There is no table for this root pointer register. The MMU will internally create an Address Translation Cache (ATC) entry for this root pointer. Address translation using this type of root pointer proceeds as follows: The logical address is compared to the limit field. If it is in range, the sum of the logical address and bits 31–4 of the root pointer comprise the physical address. Note that this is essentially a single-segment base/bounds MMU configuration.
$2	The table whose address is in bits 31–4 consists of valid short format descriptors. These descriptors are 4 bytes in length and are described later in this chapter.
$3	The table whose address is in bits 31-4 consists of valid long format descriptors. Long format descriptors are 8 bytes in length.

For descriptor types 2 and 3, the table is indexed either by the function codes FC2–FC0 or by the address bits selected by TIA in the TC register. If the Function Code Lookup (FCL) bit in the TC register is set, FC2–FC0 form the index. Otherwise, the logical address bits selected by TIA are used to index the table. As noted above, the DMA Root Pointer is selected by bus cycles run with bit FC3 equal to one. This is true regardless of the setting of the FCL bit.

[68851 Only] Root Pointer Table The 68851 remembers the last eight root pointer values for the CPU Root Pointer (CRP) register. Descriptors in the Address Translation Cache (ATC) are marked with the ID's of those remembered values. IDs range from 0–7. If the CRP is reloaded using one of the values in this Root Pointer Table (RPT), any matching descriptors can simply be reused. This feature improves performance on multitasking systems that require a great deal of switching between tasks.

[68030 only] Transparent Translation Registers Registers TT0 and TT1 define two areas of logical address space that are used directly as physical addresses, without translation. Each of these areas must be at least 16Mb (24 address bits) in size. Accesses within a transparent translation region incur no translation overhead. The transparent translation registers have the format shown in Figure 5-10.

Bits 31–24 indicate which 16Mb region is to be treated transparently. The Logical Address Mask field (Bits 23–16) allows you to ignore certain of these bits. You can use this field to enlarge a transparent region to 32Mb, 64 Mb, and so forth. A one value in a bit in the Logical Address Mask causes that bit to be ignored in comparing the logical address to the Logical Address Base field.

Bit 15 (**E**) is the enable bit for either TT0 or TT1. A one value in this bit indicates that the region described by the register is valid. This enable bit is independent of the one in the Translation Control (TC) register so that the transparent translation feature can be enabled even if the rest of the MMU is not. Transparent translation is also unaffected by the MMUDIS signal, which disables the 68030 MMU at the request of external hardware.

Bit 10 (**CI**) is the Cache Inhibit bit. Setting this bit prevents transparent region accesses from using the code or data caches on the 68030. This is useful for memory areas that are shared between two processors, for instance. Note that this feature may be useful in systems that do not require the MMU for any other reason.

Bit 9 (**R/W**) allows you to select only read cycles or only write cycles for translation. A one in this bit means only read cycles are translated, and a zero means only write cycles. To get both types translated, use bit 8.

Bit 8 (**RWM**) controls whether the value of bit 9 has any effect on translation. If this bit is clear, then the R/W field (bit 9) is used. A one value in the RWM bit causes the setting of the R/W bit to be ignored. If bit 8 is set, all cycles are translated. Setting bit 8 is required for proper operation of read/modify/write memory cycles in the transparent translation region.

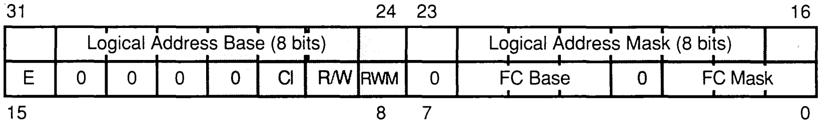


Figure 5-10. Transparent Translation Register Layout

Bits 6–4 (**FC Base**) are used to regulate the kinds of cycles that qualify for transparent translation. This field contains the desired value of the function code bits FC2–FC0 for a translated access.

Bits 2-0 (**FC Mask**) allow you to specify "Don't Care Bits" for the function code comparison. A one in any of these bits causes that bit to be ignored in the function code comparison. For example, to translate both User Data (function code 1) and Supervisor Data (function code 5) references, you would specify a value of $41 in the low-order byte of the register. (A mask value of 4 and a comparison value of 1 will match either 1 or 5, but no others.)

[68851 Only] Breakpoint Registers The 68851 MMU contains a set of eight breakpoint registers which allow you to set a count value for each of the eight BKPT instruction opcodes ($4848 - $484F). There are two registers associated with each breakpoint: a data register and a control register. These are called the Breakpoint Acknowledge Data Register (BADR0–BADR7), and the Breakpoint Acknowledge Control Register (BACR0–BACR7), respectively. There are eight such pairs with the format shown in Figure 5-11.

The Breakpoint Acknowledge Control Register (BACRx) contains an enable bit (bit 15) and an 8-bit skip count. If the breakpoint is enabled (BACRx bit 15 is a one), the contents of the Breakpoint Acknowledge Data Register (BADRx) will be substituted for the corresponding breakpoint opcode. The skip count indicates the number of times (0–255) this substitution will be performed before a breakpoint exception is taken.

The reasoning behind this feature is to allow execution profiling in near real-time. A breakpoint substitution causes a Breakpoint Acknowledge bus cycle to run, thereby allowing the substitution of an opcode without taking an exception. Since this is a very low-overhead operation, the execution timing is very close to running the same software without breakpoints. For certain real-time applications, this timing similarity is crucial.

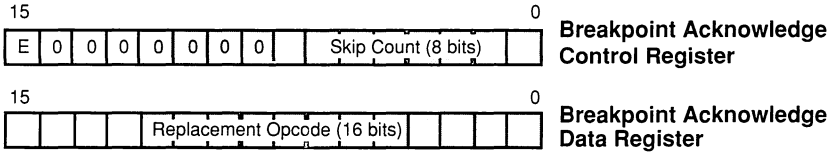

15
0
E
0
0
0
0
0
0
0
Skip Count (8 bits)
Breakpoint Acknowledge Control Register
15
0
Replacement Opcode (16 bits)
Breakpoint Acknowledge Data Register

Figure 5-11. 68851 Breakpoint Registers

[68851 Only] CAL, VAL, SCC and AC Access Control Registers The 68851 supports the 68020 CALLM/RETM instructions and the associated concept of fine-grained memory protection. There are four registers associated with this feature: the Current Access Level (CAL), Validate Access Level (VAL), Stack Change Control (SCC), and the Access Control (AC) registers.

Memory protection is accomplished using a small number of upper bits of the logical address. These bits correspond to the access level at or above which the address may be used. You can use from zero to three of these bits to achieve zero to eight layers of protection.

The Current Access Level and Validate Access Level registers give the present and previous values of the access level, respectively. These registers have the layout shown in Figure 5-12.

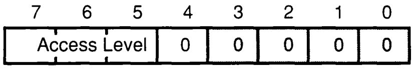

7
6
5
4
3
2
1
0
Access Level
0
0
0
0
0

Figure 5-12. 68851 CAL and VAL Register Layout

The access level is only 3 bits, located in bits 7–5 of the registers. Access levels range from 0 (highest privilege) to 7 (lowest privilege). The other bits read as zero and are ignored on write. The VAL (previous) register is copied automatically from the CAL (current) register on CALLM. The CAL register is loaded from the module descriptor on CALLM and RETM. Both registers can be changed using PMOVE or PRESTORE instructions.

The Stack Change Control (SCC) register indicates the points at which a stack change is required. There are eight privilege levels, 0–7. The SCC register contains 1 bit per level. Changing from a lower level to a higher level causes a stack change if any of the corresponding bits in the SCC register are set. The range is inclusive. For example, changing from level 7 to level 5 will cause a stack change if the SCC has any value in the range $20–$E0 (either bit 5, 6 or 7 is set). This register has the layout shown in Figure 5-13.

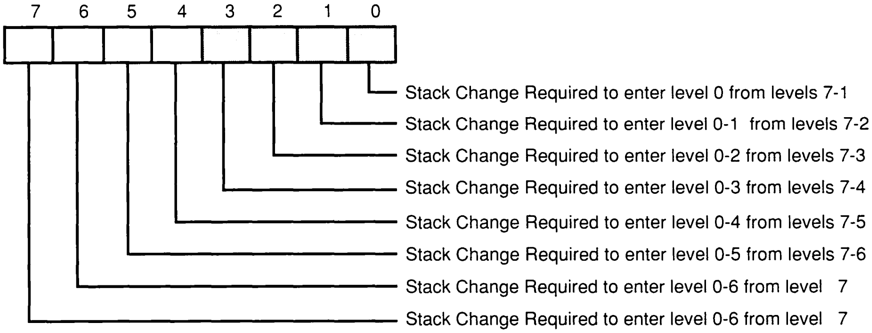

7
6
5
4
3
2
1
0
Stack Change Required to enter level 0 from levels 7-1
Stack Change Required to enter level 0-1 from levels 7-2
Stack Change Required to enter level 0-2 from levels 7-3
Stack Change Required to enter level 0-3 from levels 7-4
Stack Change Required to enter level 0-4 from levels 7-5
Stack Change Required to enter level 0-5 from levels 7-6
Stack Change Required to enter level 0-6 from level 7
Stack Change Required to enter level 0-6 from level 7

Figure 5-13. 68851 SCC Register Layout

The Access Control (AC) register specifies the options associated with the 68851 access control feature. This register has the layout shown in Figure 5-14.

Bit 7 (**MC**) is the Module Control bit. This bit determines whether or not the module feature is enabled. A value of one means the module mechanism is in effect. A value of zero disables the module feature, and all CALLM, RETM, and PVALID instructions will cause an exception.

Bits 5–4 (**ALC**) are the Access Level Control field. The setting of these bits has meaning only when the MC bit is a one. This field is initialized to zero on reset and can have the following values:

ALC Field	Meaning
$0	Access level checking is disabled, and no address bits are used for this feature.
$1	Access level checking is enabled, and one address bit (A31) is used to give two access levels.
$2	Access level checking is enabled, and two address bits (A31-A30) are used to yield four access levels.
$3	Access level checking is enabled, and three address bits (A31-A29) are used to give eight access levels.

Bits 1–0 (**MDS**) contain the Module Descriptor Size. These bits control the allowable boundary for modules to be used by the CALLM instruction. Allowable values are

MDS Field	Meaning
$0	All module descriptors are invalid.
$1	Valid module descriptors are aligned on 16-byte boundaries (A3–A0 are 0).
$2	Valid module descriptors are aligned on 32-byte boundaries (A4–A0 are 0).
$3	Valid module descriptors are aligned on 64-byte boundaries (A5–A0 are 0).

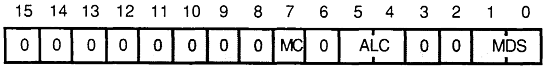


Figure 5-14. 68851 Access Control (AC) Register

[68851 Only] Cache Status Register The 68851 has a register that indicates the internal status of the Address Translation Cache (ATC). This register has the format shown in Figure 5–15.

Bit 15 (**F**) is the Flush bit. This bit is set to indicate that ATC entries from the task root pointer associated with the Task Alias field has been flushed. This is usually the result of reloading the CPU Root Pointer (CRP).

Bit 14 (**LW**) is the Lock Warning bit. This bit is set to indicate that all entries except one in the ATC have been locked. (A locked entry is one in which the descriptor has the lock (L) bit set. This feature is explained below.) The 68851 refuses to lock all entries because the need for a new entry would mean deadlock.

Bits 2–1 (**Task Alias**) contain the internal alias that the 68851 assigns to the address associated with the CPU Root Pointer (CRP) register. The 68851 can keep track of up to eight of these, thus reducing the amount of time in switching between tasks (*context switching*) in a multitasking system. This cache of root pointer values is called the Root Pointer Table (RPT), as described previously.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F	LW	0	0	0	0	0	0	0	0	0	0	0	Task Alias		

Figure 5-15. 68851 Cache Status Register (PCSR) Format

Translation Table Descriptors

Entries in a translation table are called *descriptors*. These descriptors take many forms and can be either 4 or 8 bytes in length (*short format* and *long format*, respectively).

Another attribute of a translation table descriptor is whether or not the translation descriptor is part of an intermediate table or the lowest level table. Descriptors that are part of the lowest level table are called *page descriptors*. Elements of intermediate tables are called *table descriptors*.

There are three other types of descriptors. You can use an *invalid descriptor* to eliminate either a page in a page table or a subordinate table in an higher level table. This effectively discards a portion of the logical address space. A page descriptor (*indirect descriptor*) can reference a descriptor in another table. This feature is useful for sharing a single memory area between programs in different address spaces. Placing a page descriptor in an intermediate table yields a feature called *early termination*, which allows you to map a contiguous area of physical memory with a single descriptor.

Table Descriptors Table descriptors can be either long or short format. The long formats differ between the 68030 and 68851, as shown in Figure 5-16.

The next level table address is contained in bits 31–4 in all formats. The Descriptor Type (**DT**) field in bits 1–0 of the short format and bits 33–32 of both long formats gives the type of descriptor. This field is 10 for short format and 11 for long format, as shown above.

The **WP** (Write Protect) bit (2 in short format, 34 in long format) allows you to write protect an entire range of the address space. This bit is one to protect the subordinate translation table and zero to allow writing. Note that you can individually protect elements of the lower level table if WP of an upper level table is zero. Write Protection in an upper level table entry means that all addresses governed by that entry cannot be modified.

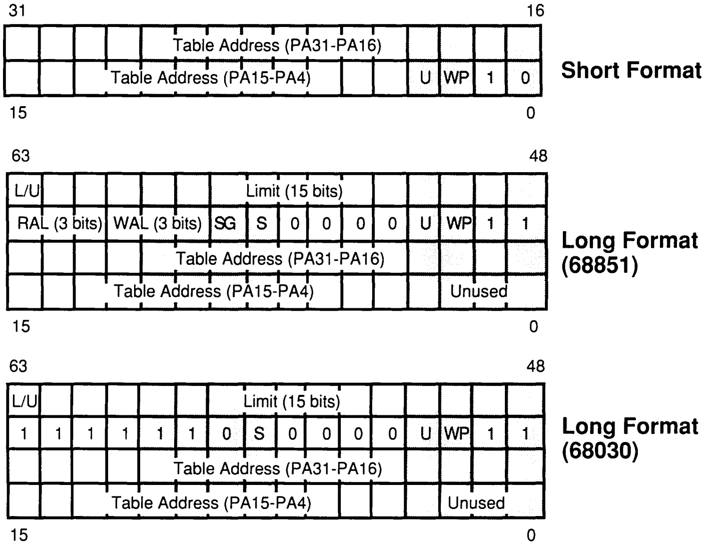

31
16
Table Address (PA31-PA16)
Table Address (PA15-PA4)
U
WP
1
0
15
0
Short Format
63
48
L/U
Limit (15 bits)
RAL (3 bits)
WAL (3 bits)
SG
S
0
0
0
0
U
WP
1
1
Table Address (PA31-PA16)
Table Address (PA15-PA4)
Unused
15
0
Long Format (68851)
63
48
L/U
Limit (15 bits)
1
1
1
1
1
1
0
S
0
0
0
0
U
WP
1
1
Table Address (PA31-PA16)
Table Address (PA15-PA4)
Unused
15
0
Long Format (68030)

Figure 5-16. Table Descriptor Formats

The **U** (Used) bit (3 in short format, 35 in long format) is set by the MMU whenever the entry is used in an address translation. This bit must be reset by software; it is never reset by the MMU. Note that this bit can be set in a higher level table, only to have the access aborted by information in a lower level table.

Long Format bit 63 (**L/U**) is the Lower/Upper bit. This bit controls the interpretation of the limit field. If this bit is a one, the root pointer contains the address of a grow-down table. All entries using this root pointer must have values greater than or equal to the limit field. When the L/U bit is a one, then all entries must have values less than or equal to the limit value. This feature is not normally used for a root pointer register, but has great value when applied to a translation table for a stack area, as described below.

Long Format bits 62-48 (**Limit**) form a 15-bit unsigned binary quantity that contains the lowest or highest legitimate value (entry number) for the table addressed by the root pointer register, as described above.

Long format bit 40 (**S**) indicates that the table mapped by the descriptor may be accessed only from Supervisor mode. The bit is a one to restrict access in this manner and zero to allow access from either supervisor or user mode.

Long format bit 41 (**SG**), when set, indicates that the translation table data is shared globally by all tasks, and that the MMU need only create one set of descriptors in the Address Translation Cache (ATC) for entries from this table. This bit is implemented only on the 68851 and must be zero on the 68030.

Long format bits 47–45 and 44–42 implement the 68851 Read Access Level and Write Access Level, respectively. These fields define the minimum access level that is allowed to read or write the memory area controlled by the descriptor. Note that on the 68030 these fields are both required to be 7, allowing access by all privilege levels. As a result, 68030 code can run on a 68020/68851 system without change.

Page Descriptors A descriptor in the lowest order table with the Descriptor Type (**DT**) field set to 01 is a page descriptor. Like table descriptors, page descriptors can have both long (8 byte) and short (4 byte) formats. The format is selected by the descriptor that contains the address of the current table, in either a higher level table or a root pointer register. These formats are shown in Figure 5-17.

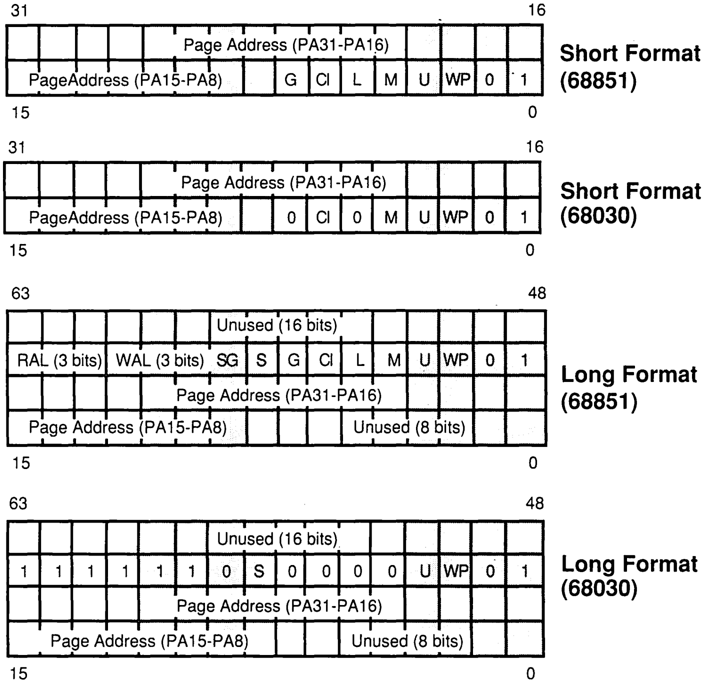

31
16
Page Address (PA31-PA16)
PageAddress (PA15-PA8)
G
CI
L
M
U
WP
0
1
15
0
Short Format (68851)
31
16
Page Address (PA31-PA16)
PageAddress (PA15-PA8)
0
CI
0
M
U
WP
0
1
15
0
Short Format (68030)
63
48
Unused (16 bits)
RAL (3 bits)
WAL (3 bits)
SG
S
G
CI
L
M
U
WP
0
1
Page Address (PA31-PA16)
Page Address (PA15-PA8)
Unused (8 bits)
15
0
Long Format (68851)
63
48
Unused (16 bits)
1 1 1 1 1 1 0 S 0 0 0 0 U WP 0 1
Page Address (PA31-PA16)
Page Address (PA15-PA8)
Unused (8 bits)
15
0
Long Format (68030)

Figure 5-17. Page Descriptor Formats

The Page Address is contained in bits 31–8 of all descriptors. Note that this address is 4 bits shorter than the address field in the Table Descriptors and Root Pointer registers. Since the minimum page size is 256 bytes, the 8 low-order bits of the address cannot be used.

The next level table address is contained in bits 31–4 in all formats. The Descriptor Type (**DT**) field in bits 1-0 of the short format and bits 33–32 of both long formats gives the type of descriptor. This field is 01 for both short and long format page descriptors, as shown above.

The (**WP**) Write Protect bit (2 in short format, 34 in long format) allows you to write protect a single memory page. This bit is one to protect the page and zero to allow writing.

The (**U**) Used bit (3 in short format, 35 in long format) is set by the MMU whenever the entry is used in an address translation. This bit must be reset by software; it is never reset by the MMU. In a virtual memory system, you can use this bit as a reference indication.

A related indicator is the **(M)** Modified bit (4 in short format, 36 in long format). This bit is also set by the MMU, but must be cleared by software. The MMU sets this bit in the descriptor on the first write cycle to the affected page. You can use this bit to avoid writing a page back to disk if it has not been changed.

The **(L)** Lock bit (5 in short format, 37 in long format) causes a descriptor to be locked into the Address Translation Cache (ATC) until this cache is flushed. This feature is implemented only on the 68851; the L bit must be zero on the 68030. There are 64 ATC entries on a 68851. The MMU allows you to lock all but one of these (to avoid deadlock).

The **(CI)** Cache Inhibit bit (6 in short format, 38 in long format) inhibits caching on the selected page. For every memory cycle referencing the selected page, an output signal is asserted, thereby indicating that external caching is to be disabled. On the 68030, the internal cache is inhibited as well.

The **(G)** Gate bit (7 in short format, 39 in long format), if set, indicates that the page can contain CALLM module descriptors used for access level translations. These module descriptors are also called "gates," and are similar to the privilege level transition mechanisms used by other manufacturers. This bit is available only on the 68851 chip.

Long format bit 40 **(S)** indicates that the page mapped by the descriptor may be accessed only from Supervisor mode. The bit is a one to restrict access in this manner, and zero to allow access from either supervisor or user mode.

Long format bit 41 **(SG)**, when set, indicates that the translation table data is shared globally by all tasks and that the MMU need only create one set of descriptors in the Address Translation Cache (ATC) for entries from this table. This bit is implemented only on the 68851 and must be zero on the 68030.

Long format bits 47–45 and 44–42 implement the 68851 Read Access Level and Write Access Level, respectively. These fields define the minimum access level that is allowed to read or write the memory area controlled by the descriptor. Note that on the 68030, these fields are both required to be 7, thereby allowing access by all privilege levels. 68030 code can run on a 68020/68851 system without change as a result.

Long format bits 63–48 and 7–0 are unused by the MMU and can be used by software for page status information. If the page size is greater than 256 bytes, you may also use the low-order address bits that are not required for the page size you have selected.

Invalid Descriptors An invalid descriptor is any descriptor that has 00 in the descriptor type (**DT**) field (bits 1–0 in short format, 33–32 in long format). All other bits in the descriptor are unused and may be used for any purpose. Invalid descriptors appear in Figure 5-18.

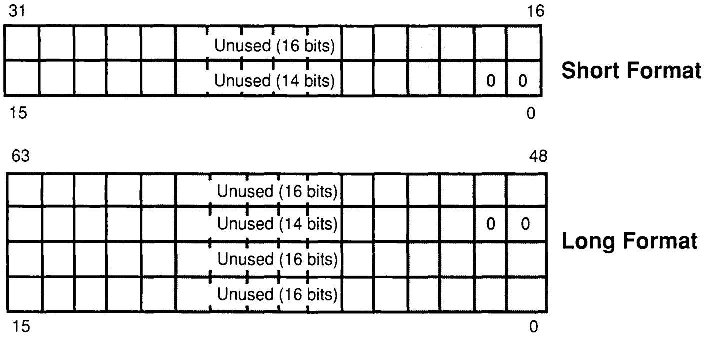

31
16
Unused (16 bits)
Unused (14 bits)
0
0
15
0
Short Format
63
48
Unused (16 bits)
Unused (14 bits)
0
0
Unused (16 bits)
Unused (16 bits)
15
0
Long Format

Figure 5-18.4 Invalid Descriptor Formats

Invalid descriptors are required for virtual memory; they indicate pages on disk instead of in primary memory. Upon encountering a reference to an invalid descriptor, the processor takes a BUSERR exception. You can then load the page from disk, change the descriptor to a valid descriptor, and continue the faulted program. An invalid descriptor may occur at any level of the translation table except the root pointer register.

Indirect Descriptors An indirect descriptor is a table descriptor where a page descriptor should be. The descriptor type is either 10 or 11, indicating a 4- or 8-byte format. Note that these types are also used for the table descriptors described above. Bits 31–2 contain the address of a page descriptor, as shown in Figure 5-19.

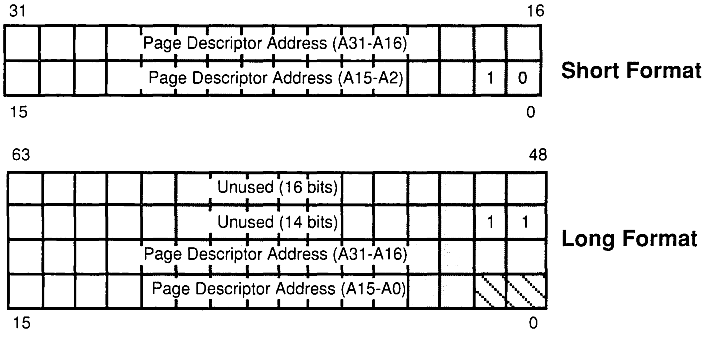
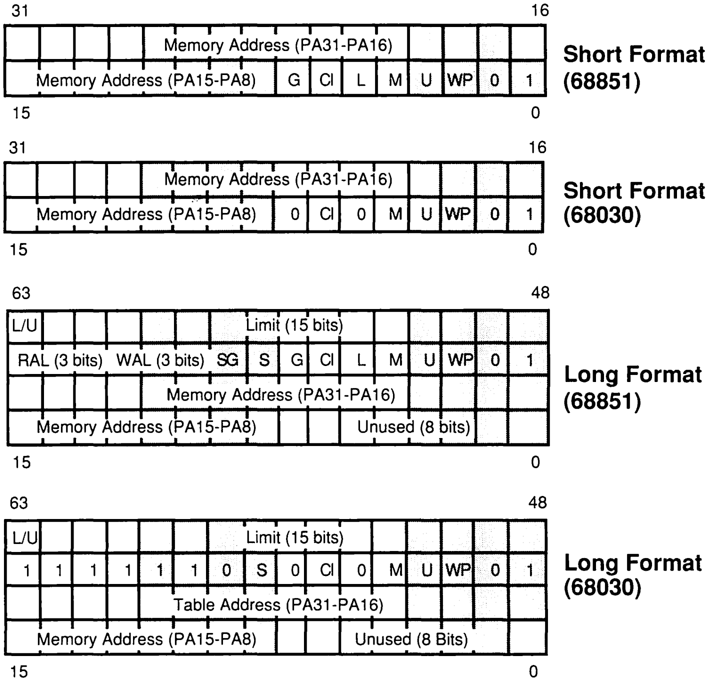

31
16
Page Descriptor Address (A31-A16)
Page Descriptor Address (A15-A2)
1
0
15
0
Short Format
63
48
Unused (16 bits)
Unused (14 bits)
1
1
Page Descriptor Address (A31-A16)
Page Descriptor Address (A15-A0)
15
0
Long Format

Figure 5-19. Indirect Descriptor Formats

A short format indirect descriptor must contain the address of a short format page descriptor. A long format indirect descriptor must contain the address of a long format page descriptor. Long format bits 63–34 and 2–0 are unused.

Early Termination Descriptors An early termination descriptor is a page descriptor where a table descriptor should be. The descriptor type is 01. The format is selected by the higher level table, except for a root pointer. Root pointers are constrained to long format. Early termination formats are shown in Figure 5-20.

A short format early termination descriptor replaces a lower-level table with an area of contiguous physical memory starting at the memory address in the descriptor. A long format also performs this function, with the addition of direction and limit fields. Note that this feature allows you to implement base/bounds memory management in combination with paged memory management. You can implement demand segmentation similar to that used on the Intel 80286 chip by the OS/2 operating system.



31
16
Memory Address (PA31-PA16)
Memory Address (PA15-PA8)
G
CI
L
M
U
WP
0
1
15
0
Short Format (68851)
31
16
Memory Address (PA31-PA16)
Memory Address (PA15-PA8)
0
CI
0
M
U
WP
0
1
15
0
Short Format (68030)
63
48
L/U
Limit (15 bits)
RAL (3 bits)
WAL (3 bits)
SG
S
G
CI
L
M
U
WP
0
1
Memory Address (PA31-PA16)
Memory Address (PA15-PA8)
Unused (8 bits)
15
0
Long Format (68851)
63
48
L/U
Limit (15 bits)
1 1 1 1 1 1 0 S 0 CI 0 M U WP 0 1
Table Address (PA31-PA16)
Memory Address (PA15-PA8)
Unused (8 Bits)
15
0
Long Format (68030)

Figure 5-20. Early Termination Descriptor Formats

The **(WP)** Write Protect bit (2 in short format, 34 in long format) allows you to write-protect the memory described by the descriptor.

The **(U)** Used bit (3 in short format, 35 in long format) is set by the MMU whenever the entry is used in an address translation. This bit must be reset by software; it is never reset by the MMU. In a virtual memory system, you can use this bit as a reference indication.

A related indicator is the **(M)** Modified bit (4 in short format, 36 in long format). This bit also is set by the MMU, but must be cleared by software. The MMU sets this bit in the descriptor on the first write cycle to the affected page. You can use this bit to avoid writing a region back to disk if it has not been changed.

The **(L)** Lock bit (5 in short format, 37 in long format) causes a descriptor to be locked into the Address Translation Cache (ATC) until this cache is flushed. This feature is implemented only on the 68851; the L bit must be zero on the 68030. There are 64 ATC entries on a 68851. The MMU allows you to lock all but one of these (to avoid deadlock).

The **(CI)** Cache Inhibit bit (6 in short format, 38 in long format) inhibits caching on the selected memory area. For every memory cycle referencing the selected page, an output signal is asserted, indicating that external caching is to be disabled. On the 68030, the internal cache is inhibited as well.

The **(G)** Gate bit (7 in short format, 39 in long format), if set, indicates that the region can contain CALLM module descriptors used for access level translations. These module descriptors are also called "gates," and are similar to the privilege level transition mechanisms used by other manufacturers. This bit is available only on the 68851 chip.

Long format bit 40 **(S)** indicates that the region mapped by the descriptor may be accessed only from Supervisor mode. The bit is a one to restrict access in this manner and zero to allow access from either supervisor or user mode.

Long format bit 41 **(SG)**, when set, indicates that the translation table data is shared globally by all tasks and that the MMU need only create one descriptor in the Address Translation Cache (ATC) for this region. This bit is implemented only on the 68851, and must be zero on the 68030.

Long format bits 47–45 and 44–42 implement the 68851 Read Access Level and Write Access Level, respectively. These fields define the minimum access level that is allowed to read or write the memory area controlled by the descriptor. Note that on the 68030 these fields are both required to be 7, allowing access by all privilege levels. As a result, 68030 code can run on a 68020/68851 system without change.

Bit 63 (**L/U**) is the Lower/Upper bit. This bit controls the interpretation of the limit field. If this bit is a one, the descriptor contains the address of a grow-down table. All entries using this descriptor must have values greater than or equal to the limit field. When the L/U bit is a one, all entries must have values less than or equal to the limit value. The grow-down area is most useful for stack regions that need to expand toward lower addresses.

Bits 62–48 (**Limit**) is a 15-bit unsigned binary quantity that contains the lowest or highest legitimate value for the memory area addressed by the descriptor, as described above.

Long format bits 7-0 are unused by the MMU and can be used by software for page status information. If the page size is greater than 256 bytes, you may also use the low-order address bits that are not required for the page size you have selected.

[66851 Only]

MMU Conditional Operations

The 68851 implements conditional branch (PBcc) and conditional loop (PDBcc) instructions. These instructions are similar to the Bcc and DBcc main processor instructions, except that the conditions are evaluated by the coprocessor. These instructions are implemented only on the 68851 MMU.

MMU conditional operations allow interrogation of the bits in the MMUSR. You can interrogate only one of these bits at a time, as shown below:

Condition	Mnemonic	Meaning
000000	BS -- B set	BUSERR indicator bit (15) in MMUSR is set
000001	BC -- B clear	BUSERR indicator bit (15) in MMUSR is clear
000010	LS -- L set	Lock indicator bit (14) in MMUSR is set
000011	LC -- L clear	Lock indicator bit (14) in MMUSR is clear
000100	SS -- S set	Supervisor indicator bit (13) in MMUSR is set
000101	SC -- S clear	Supervisor indicator bit (13) in MMUSR is clear
000110	AS -- A set	Access violation indicator bit (12) in MMUSR is set
000111	AC -- A clear	Access violation indicator bit (12) in MMUSR is clear
001000	WS -- W set	Write protection indicator bit (11) in MMUSR is set
001001	WC -- W clear	Write protection indicator bit (11) in MMUSR is clear

001010	IS -- I set	Invalid descriptor indicator bit (10) in MMUSR is set
001101	IC -- I clear	Invalid descriptor indicator bit (10) in MMUSR is clear
001110	GS -- G set	Gate indicator bit (9) in MMUSR is set
001111	GC -- G clear	Gate indicator bit (9) in MMUSR is clear

The condition field is the 6-bit field in the coprocessor conditional instructions.

[68851 Only]

MMU State Frames

The PSAVE and PRESTORE instructions allow storing the internal MMU state in memory and retrieving it for later use. These instructions are similar to the FSAVE and FRESTORE for the 68881/2 floating point chips. The 68030 does not support the PSAVE and PRESTORE functions and thus has no associated state frame information.

There are four types of state frame for a 68851: Reset, Idle, Mid-Coprocessor, and Breakpoint Enabled. Each of these state frames uses a different format memory area for the PSAVE and PRESTORE instructions. Each of these formats is discussed below.

Reset Format State Frame The 68851 supports the notion of restoring a null state frame to initialize the chip. This feature is also found in the 68881/2 floating point chips. The format of the state frame is similar, and appears in Figure 5-21.

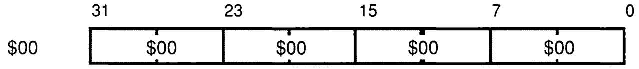
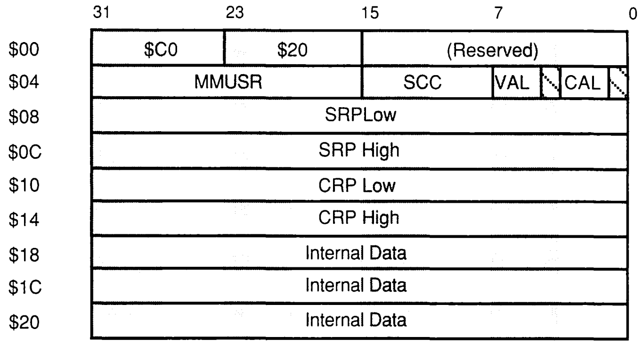


Figure 5-21. Reset Format State Frame

This state frame is just a longword of binary zeros. The 68881/2 chips use a compatible format to generate an initialization with the coprocessor restore instruction. This state frame is also called the "Null" state frame.

Idle Format State Frame The idle format frame is generated by a PSAVE instruction whenever there is no coprocessor instruction executing and no breakpoints are enabled. This frame has the format shown in Figure 5-22.

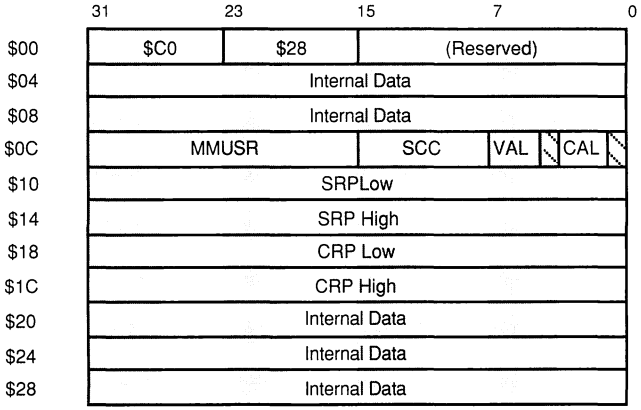


	31	23	15	7 0
$00	$C0	$20	(Reserved)	
$04	MMUSR		SCC	VAL ▨ CAL ▨
$08	SRPLow			
$0C	SRP High			
$10	CRP Low			
$14	CRP High			
$18	Internal Data			
$1C	Internal Data			
$20	Internal Data			

Figure 5-22. Idle Format State Frame

The first word gives the version number in bits 31–24, and the length of the frame (not including the first longword) in bits 23–16. The state frame contains the MMUSR, SCC, VAL, CAL, SRP and CRP registers. The three longwords of internal data have unpublished contents and should not be modified.

Mid-Coprocessor Format State Frame The PSAVE instruction generates a mid-coprocessor state frame when a coprocessor instruction is in progress and the breakpoint facility is not in use. This state frame is shown in Figure 5-23.



	31	23	15	7 0
$00	$C0	$28	(Reserved)	
$04	Internal Data			
$08	Internal Data			
$0C	MMUSR		SCC	VAL ▨ CAL ▨
$10	SRPLow			
$14	SRP High			
$18	CRP Low			
$1C	CRP High			
$20	Internal Data			
$24	Internal Data			
$28	Internal Data			

Figure 5-23. Mid-Coprocessor Format State Frame

The only difference between the Mid-Coprocessor and Idle formats is the insertion of eight bytes of internal data at offset 4.

Breakpoint Enabled Format State Frame Whenever the 68851 breakpoint feature is enabled, then PSAVE generates the format shown in Figure 5-24.

This is a Mid-Coprocessor Format state frame with the breakpoint control registers inserted at offset 4.

$00	$C0	$48	(Reserved)			
$04	BAC0		BAD0			
$08	BAC1		BAD1			
$0C	BAC2		BAD2			
$10	BAC3		BAD3			
$14	BAC4		BAD4			
$18	BAC5		BAD5			
$1C	BAC6		BAD6			
$20	BAC7		BAD7			
$24	Internal Data					
$28	Internal Data					
$2C	MMUSR		SCC	VAL	▨	CAL ▨
$30	SRPLow					
$34	SRP High					
$38	CRP Low					
$3C	CRP High					
$40	Internal Data					
$44	Internal Data					
$48	Internal Data					

Figure 5-24. Breakpoint Enabled Format State Frame

MMU-Generated Exceptions

Both the 68851 and 68030 MMUs can generate several exceptions.

Name	Vector	Format	Meaning
BUSERR	04 ($08)	$A	Illegal memory reference, missing page, etc.
F-Line Emulator	11 ($2C)	$0	Invalid MMU instruction
MMU Configuration	56 ($DC)	$2	Illegal MMU configuration
MMU Illegal Operation	57 ($E0)	$2	Invalid operation (TC Enable bit = 0; 68851 only)
MMU Access Level	58 ($E4)	$2	Access violation (68851 only)

BUSERR Exception The 68851 and 68030 use the BUSERR exception for many purposes. Upon receipt of a BUSERR, the exception handler should issue a PTEST instruction for the faulting address. Bits 15–10 of the MMUSR after the PTEST indicate the type of error. Note that a 68851 can get an error from an external device using the DMA Root Pointer (DRP). You must obtain the faulting address from the device causing the fault in this case.

The BUSERR bit (B) of the MMUSR indicates that the BUSERR occurred during a main memory access. This situation indicates either an invalid pointer in the translation table or a memory failure. Note that memory failures are sometimes transient and that the B bit may not be set as a result of the PTEST operation.

If a limit violation occurs, the (**L**) Limit bit is set. Note that this is not necessarily an error in the case of stack areas that expand automatically. For most other situations, an invalid memory access is indicated. The action required depends on the system. For example, in UNIX the application can use the signal() function to handle this type of fault.

The (**S**) Supervisor bit in the MMUSR indicates that a user mode program attempted to access a memory area marked as supervisor only. This is usually a programming error.

On a 68851, access level violation occurs if the access level bits of a logical address indicate a higher privilege (lower numeric value) than that contained in the Current Access Level (CAL) register.

The MMUSR WP (Write Protect) bit indicates an attempted write into an area of memory marked as read only. On both the 68851 and 68030, this error can occur if the currently executing program has attempted to write to an area mapped by a descriptor with the WP bit set. On the 68851, this error can also occur if the Write Access Level field in the descriptor is less than the Current Access Level (CAL) register, indicating an access violation.

Line F Emulator Exception An MMU instruction that is not recognized by the coprocessor causes a Line F exception. This can include invalid addressing mode/instruction combinations, invalid bit combinations in the instruction word, or illegal values in one of the instruction fields. The Line F Emulator exception uses vector 11, with a format 0 stack frame, as shown in Figure 2-34. Note that the 68030 will generate this exception on a PLOAD instruction with a zero length parameter with an address register specification (see the PLOAD instruction description).

MMU Configuration Exception The MMU checks values loaded into the MMU registers for validity during the instruction that does the load. If the value is not proper, the MMU does not change the register contents and takes an MMU Configuration Exception through vector 56, using a format 2 stack frame. The following errors can cause an MMU Configuration Exception:

1. Loading the Translation Control Register with a value such that the sum of the PS, IS, TIA, TIB, TIC, and TID fields is not equal to 32. These fields regulate how the 32-bit address is divided. The sum must be 32.
2. Loading a value into the Translation Control Register PS (Page Size) field with bit 3 of this field equal to zero. This is an attempt to use page sizes less than 256 bytes and is illegal.
3. Loading an invalid descriptor type (00) value into any of the root pointer registers: CRP, DRP, or SRP. In this case, the register is loaded with the new (invalid) value.
4. Issuing a PRESTORE instruction with an invalid state frame format or contents.

This type of error is quite common when writing new software which uses the MMU.

[68851 Only] MMU Illegal Operation Attempting the PLOAD or PTEST MMU instructions with the MMU disabled (the E bit in the TC register is clear) causes an MMU Illegal Operation exception through vector 57 using a format 2 stack frame. The 68030 does not exhibit this behavior.

[68851 Only] MMU Access Violation A read or a write operation with the 68851 Module control feature enabled can cause an exception if the access level restrictions are violated. This exception uses vector 58 with a format 2 stack frame.

Instruction Description Format

The remainder of this chapter is dedicated to descriptions of individual MMU instructions. Each item is described below. These instructions conform to the F-line coprocessor standard described in Chapter 2. The coprocessor ID (bits 11-9 of the first instruction) is zero for the MMU. We will treat this field as a constant.

Header

The header for each instruction contains the name, mnemonic expansion, and assembly language syntax, as in the example below (from the PBcc instruction):

PBcc Instruction (Privileged) **Syntax: PBcc.W <label>**
Branch on MMU Condition (68851 Only) **PBcc.L <label>**

Although all MMU instructions are privileged, the notation in the header is retained for compatibility with the descriptions in chapters 3 and 4. When an instruction is implemented only on the 68851, that fact is noted in the second line of the header.

Description

The next area of the instruction explanation contains a prose description of the instruction and a chart showing the effects on the MMUSR, allowed data sizes, and the MMUs that support the instruction. The chart for the PBcc instruction is shown below.

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	-	-	-	-	-	-	-	-	-	-			•	•			•	

B Not affected.
L Not affected. (68851 only).
S Not affected.
A Not affected. (68851 only).
W Not affected.
I Not affected.
M Not affected.
G Not affected. (68851 only).
C Not affected. (68851 only).
N Not affected.

The MMUSR bits that are implemented only on the 68851 are duly noted.

Addressing Modes

Most MMU instructions have an effective address operand. A chart similar to the ones in Chapters 3 and 4 details which of the 68020/30 addressing modes can be used with the instruction. The chart shown below is from the PVALID instruction.

Syntax	Mode	Reg	Valid
Dn	000	Dn	
An	001	An	
(An)	010	An	•
(An)+	011	An	
-(An)	100	An	
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	
(d16,PC)	111	010	
(d8,PC,Xn)	111	011	
(bd,PC,Xn)	111	011	
([bd,PC,Xn],od)	111	011	
([bd,PC],Xn,od)	111	011	

Instruction Format

As in chapters 3 and 4, a figure details the layout of the bits in the instruction words, as shown below. This example is from the PLOAD instruction description.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
0	0	1	0	0	0	R/W	0	0	0	0	FC				

The last part of the instruction explanation is an example instruction execution. This example is from the PRESTORE instruction. Note the extra information from the debugger to describe MMU registers.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	1	Mode			Register		
1	1	1	1	0	0	0	1	0	1	0	1	0	0	0	0

The hex value is $F150. Executing the instruction yields:

Before:

```
USP=003b075a MSP=003b075a ISP =003b0758 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 003b1a68 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 003b0758
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00c064a7
TC =80f84500 PCSR = 0007 MMUSR=ffff BLSAWIMGCT--- Levels = 7
CAL=e0 VAL=e0 SCC=aa AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PRESTORE$   PRESTORE   (A0)
```

Register A0 points to an area with the following format:

```
03b1a68:      c0200000 0000ffee 40800050 7fff0202 . ......@..P ...
03b1a78:      40800050 7fff0202 ffff4200 010f0800 @..P .....B.....
03b1a88:      80f84500 00000000 00000000 00000000 ..E.............
```

This is an IDLE format state frame, as described in Figure 5-22.

After:

```
USP=003b075a MSP=003b075a ISP =003b0758 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 003b1a68 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 003b0758
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00c064a7
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=e0 VAL=e0 SCC=aa AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PRESTORE$+002          NOP
```

Note that the TC, CRP, SRP, CAL, VAL, and SCC registers were reloaded with their original contents.

All of these instructions were executed on a Macintosh II (68020) with a 68851 installed. This allowed the execution of all MMU instructions. The environment is fairly limited since the debugger runs on the Macintosh II and we had to use the Macintosh-supplied translation tables.

The translation table resides in the Macintosh ROM and simply maps addresses of the form $s00000 into $Fss00000. This is the Macintosh 24-bit mode, as described in *Inside Macintosh* Volume 5. Note that the TC register specifies a page size of 32K, an initial shift count of 8 (to discard the top 8 bits of the address), and indices of 4 bits for level A and 5 bits for level B. The 16 values in the A level table are used to index the 16 possible NuBus slots, with early termination descriptors to provide the necessary mapping to the addresses specified by *Inside Macintosh*.

The actual instruction sets for this chapter start on the next page.

PBcc Instruction (Privileged)

Branch on MMU Condition (68851 Only)

Syntax: PBcc.W <label>
PBcc.L <label>

Description

The PBcc instruction is a conditional branch, based on a condition evaluated by the MMU rather than the main processor.

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	-	-	-	-	-	-	-	-	-	-			•	•			•	

B Not affected.
L Not affected. (68851 only).
S Not affected.
A Not affected. (68851 only).
W Not affected.
I Not affected.
M Not affected.
G Not affected. (68851 only).
C Not affected. (68851 only).
N Not affected.

If the MMUSR bit specified by the condition field in the instruction (see below) is in the state specified (clear or set), a branch takes place. Otherwise, execution continues with the next sequential instruction.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	1	Size	MMU Condition					
16-bit displacement if Size = 0															
32-bit displacement Size = 1															

Bit 6 is zero for a 16-bit displacement and one for a 32-bit displacement. As with other 68000 family conditional branches, the value of the PC used in computing the new address is the address of the first displacement word (the instruction address plus two).

The MMU Condition allows you to check the state of a single MMUSR bit. Values for this field are as follows:

Condition	Mnemonic	Meaning
000000	BS -- B set	BUSERR indicator bit (15) in MMUSR is set
000001	BC -- B clear	BUSERR indicator bit (15) in MMUSR is clear
000010	LS -- L set	Lock indicator bit (14) in MMUSR is set
000011	LC -- L clear	Lock indicator bit (14) in MMUSR is clear
000100	SS -- S set	Supervisor indicator bit (13) in MMUSR is set
000101	SC -- S clear	Supervisor indicator bit (13) in MMUSR is clear
000110	AS -- A set	Access violation indicator bit (12) in MMUSR is set
000111	AC -- A clear	Access violation indicator bit (12) in MMUSR is clear
001000	WS -- W set	Write protection indicator bit (11) in MMUSR is set
001001	WC -- W clear	Write protection indicator bit (11) in MMUSR is clear
001010	IS -- I set	Invalid descriptor indicator bit (10) in MMUSR is set
001101	IC -- I clear	Invalid descriptor indicator bit (10) in MMUSR is clear
001110	GS -- G set	Gate indicator bit (9) in MMUSR is set
001111	GC -- G clear	Gate indicator bit (9) in MMUSR is clear

Example The PBBC *+$20 instruction branches to offset $20 from the location of the first word of the instruction. The machine code has the following layout:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>Size</td><td colspan="6">MMU Condition</td></tr>
<tr><td colspan="16">16-bit displacement if Size = 0</td></tr>
<tr><td colspan="16">32-bit displacement if Size = 1</td></tr>
</table>

1	1	1	1	0	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0

The hex value is $F081001E. Executing the instruction yields:

Before:

```
USP=003b09f6 MSP=003b09f6 ISP =003b09f4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 003b09f4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00c064a7
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=e0 VAL=e0 SCC=46 AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PBCC$       PBBC        *+$0020       ; PBCC$+020
```

After:

```
USP=003b09f6 MSP=003b09f6 ISP =003b09f4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 003b09f4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00c064a7
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=e0 VAL=e0 SCC=46 AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PBCC$+020   NOP
```

In this example, the PPBC (B Clear) branch is taken because the MMUSR B bit (bit 15) is clear.

PDBcc Instruction (Privileged); Syntax: PDBcc Dn,<label>

Test MMU Condition, Decrement, and Branch (68851 Only)

Description

The PDBcc instruction evaluates an MMU condition combination. If the specified condition is false, the low word of the data register is decremented. If the resulting value is not -1, a branch to the specified label is taken. PDBcc is normally used as the last instruction in a loop. Note that an initial value of -1 in the data register will cause 65,536 iterations (the largest possible value using DBcc) of the loop. The condition may be viewed as the termination condition of a loop, much like the do/while construct in C, or the REPEAT/UNTIL construct in the Pascal language. The PDBcc condition, unlike the language constructs, terminates the loop when the condition is true, rather than when false. This instruction is similar to the DBcc main processor instruction, except that the coprocessor evaluates the condition.

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	-	-	-	-	-	-	-	-	-	-			•				•	

B	Not affected.
L	Not affected. (68851 only).
S	Not affected.
A	Not affected. (68851 only).
W	Not affected.
I	Not affected.
M	Not affected.
G	Not affected. (68851 only).
C	Not affected. (68851 only).
N	Not affected.

If the MMUSR bit specified by the condition field in the instruction (see below) is in the state specified (clear or set), the loop terminates. Otherwise, the data register is decremented and the branch is taken if the result is not -1.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	1	0	0	1	Data Register		
0	0	0	0	0	0	0	0	0	0	MMU Condition					
16-bit displacement															

As with other 68000 family conditional branches, the value of the PC used in computing the new address is the address of the first displacement word (the instruction address plus two).

The MMU Condition allows you to check the state of a single MMUSR bit. Values for this field are as follows:

Condition	Mnemonic	Meaning
000000	BS -- B set	BUSERR indicator bit (15) in MMUSR is set.
000001	BC -- B clear	BUSERR indicator bit (15) in MMUSR is clear.
000010	LS -- L set	Lock indicator bit (14) in MMUSR is set.
000011	LC -- L clear	Lock indicator bit (14) in MMUSR is clear.
000100	SS -- S set	Supervisor indicator bit (13) in MMUSR is set.
000101	SC -- S clear	Supervisor indicator bit (13) in MMUSR is clear.
000110	AS -- A set	Access violation indicator bit (12) in MMUSR is set..
000111	AC -- A clear	Access violation indicator bit (12) in MMUSR is clear.
001000	WS -- W set	Write protection indicator bit (11) in MMUSR is set.
001001	WC -- W clear	Write protection indicator bit (11) in MMUSR is clear.
001010	IS -- I set	Invalid descriptor indicator bit (10) in MMUSR is set.
001101	IC -- I clear	Invalid descriptor indicator bit (10) in MMUSR is clear.
001110	GS -- G set	Gate indicator bit (9) in MMUSR is set.
001111	GC -- G clear	Gate indicator bit (9) in MMUSR is clear.

Example The PDBBC D0,*-$16 instruction branches to offset negative $16 from the location of the instruction. The machine code has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	1	0	0	1	Data Register		
0	0	0	0	0	0	0	0	0	0	MMU Condition					
16-bit displacement															

1	1	1	1	0	0	0	0	0	1	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0

The hex value is $F048 0001 FFE6. Executing the instruction yields:

Before:

```
USP=003b0a42 MSP=003b0a42 ISP =003b0a40 SR =2700 -- -- SI PL=7 -----
D0: 00000000 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 003b0a40
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=001df10a
TC =80f84500 PCSR = 0000 MMUSR=8000 B------------ Levels = 0
CAL=e0 VAL=e0 SCC=a6 AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PDBCC$      PDBBC      D0,*-$0016    ; PDBTARGET
```

After:

```
USP=003b0a42 MSP=003b0a42 ISP =003b0a40 SR =2700 -- -- SI PL=7 -----
D0: 0000ffff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 003b0a40
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=001df10a
TC =80f84500 PCSR = 0000 MMUSR=8000 B------------ Levels = 0
CAL=e0 VAL=e0 SCC=a6 AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PDBCC$+006  NOP
```

Note that the loop terminated on the count. The value of D0 was initially zero. When decremented, the value became -1 and caused the loop to terminate.

PFLUSH Instruction (Privileged)

Flush MMU ATC Entries

Syntax:PFLUSHA
PFLUSH <fc>,#mask
PFLUSH <fc>,#mask,<ea>
(68851 Only) PFLUSHS <fc>,#mask
(68851 Only) PFLUSHS <fc>,#mask,<ea>

Description

The PFLUSH instruction invalidates a set of Address Translation Cache (ATC) entries. The PFLUSHA instruction flushes the entire ATC. On the other forms, you can select either a function code mask or the combination of a function code mask and an effective address. The function code can be immediate, in a data register, in the CPU SFC register, or in the CPU DFC register. Note that the 68851 requires 4 function code bits to accommodate DMA accesses. Since the SFC and DFC registers have only three bits, you cannot flush DMA addresses by specifying these registers as the function code. The 68030 has only 3 function code bits and does not have this restriction.

Selecting ATC entries by function code mask works as follows: The instruction function code ANDed with the mask is compared to the ATC entry, also ANDed with the mask. The mask therefore selects the bits that participate in the comparison. If the optional effective address is specified, only those entries that map a page containing the address and also match the function code are flushed. Note that the same address may occur for supervisor, user, and DMA states, as well as for code and data, depending on the translation table setup.

The 68851 PFLUSHS variant flushes ATC entries with the SG (Shared Globally) bit set. The other PFLUSH forms do not affect these entries. The 68030 does not support the SG bit or the PFLUSHS derivative.

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	-	-	-	-	-	-	-	-	-	-							•	•

B	Not affected.
L	Not affected. (68851 only.)
S	Not affected.
A	Not affected. (68851 only.)
W	Not affected.
I	Not affected.
M	Not affected.
G	Not affected. (68851 only.)
C	Not affected. (68851 only.)
N	Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn	000	Dn	
An	001	An	
(An)	010	An	•
(An)+	011	An	
-(An)	100	An	
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	
(d16,PC)	111	010	
(d8,PC,Xn)	111	011	
(bd,PC,Xn)	111	011	
([bd,PC,Xn],od)	111	011	
([bd,PC],Xn,od)	111	011	

The addressing mode specifies the address that is to be flushed rather than a memory location containing the address. PFLUSH is similar to the Load Effective Address (LEA) instruction in this regard. For instance, if the address to be flushed is in a memory longword pointed to by A0, you must use the form PFLUSH [(A0)] rather than PFLUSH (A0). (The latter form would flush the page whose address is in A0 rather than the longword to which A0 points.)

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
0	0	1	Mode			0	Mask				FC				

The Mode field in bits 12–10 of the second word of the instruction selects the form of the instruction. Valid combinations are

Mode	Meaning	Instruction
001	All entries	PFLUSHA
100	By function code alone	PFLUSH <fc>,#mask
101	By function code with shared entries	PFLUSHS <fc>,#mask
110	By function code and effective address	PFLUSH <fc>,#mask,<ea>
111	By function code, effective address, with shared entries	PFLUSHS <fc>,#mask,<ea>

As noted above, the two forms of the instruction involving shared entries are implemented only on the 68851 MMU.

Bits 8–5 of the second instruction word specify the function code mask, corresponding to function code bits FC3–FC0. As noted previously, the FC3 bit (for DMA cycles) is not implemented on the 68030. If the mode field is 001 (PFLUSHA), the mask field must be 0000.

Bits 4–0 (FC) of the second instruction word identify the source of the function code value. These bits are interpreted as follows:

FC Field	Meaning
00000	Required if mode = 001 (PFLUSHA)
1DDDD	Function code is immediate; contained in bits 3–0 (DDDD)
01RRR	Function code is in data register RRR
00000	If mode not equal to 001, function code is in CPU SFC register
00001	Function code is in CPU DFC register

Example The PFLUSHA instruction flushes all ATC entries. The machine code has the following layout:

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="3">Mode</td><td colspan="3">Register</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td colspan="3">Mode</td><td>0</td><td colspan="4">Mask</td><td colspan="5">FC</td></tr>
</table>

1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0

The hex value is $F000 2400. Executing the instruction yields:

Before:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PFLUSH$     PFLUSHA
```

After:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PFLUSH$+004          NOP
```

This output shows no visible signs that the flush has occurred.

PFLUSHR Instruction (Privileged)

Syntax: PFLUSHR <ea>

Flush ATC and RPT Entries (68851 Only)

Description

The PFLUSHR instruction invalidates a set of Address Translation Cache (ATC) entries. The effective address points to a previously used value of the CPU Root Pointer (CRP) register. The 68851 Root Pointer Table is purged of any entry created by this CRP value, and any ATC values based on this root pointer (except shared entries) are also purged. This instruction is useful when a task is deleted from a multitasking system because it allows the root pointer to be reused by a newly created task.

If the Supervisor Root Pointer is not in use, do not issue the PFLUSHR until the CPU Root Pointer references a task different from the one being purged. Issuing a PFLUSHR with the CRP containing the purged entry simply reloads the ATC with the entries that will soon be invalid.

The 68030 does not support this instruction because it does not not have a Root Pointer Table.

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	-	-	-	-	-	-	-	-	-	-					•		•	

B Not affected.
L Not affected. (68851 only.)
S Not affected.
A Not affected. (68851 only.)
W Not affected.
I Not affected.
M Not affected.
G Not affected. (68851 only.)
C Not affected. (68851 only.)
N Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn	000	Dn	
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	•
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

The addressing mode specifies the address at which to find an old CPU Root Pointer value (8 bytes). This differs from the other PFLUSH instructions defined previously.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Example The PFLUSHR (A0) instruction flushes all ATC entries associated with the old value of the CRP whose address is in register A0. The machine code has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

The hex value is $F010 A000. Executing the instruction yields:

Before:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 002c756e 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PFLUSHR$    PFLUSHR    (A0)
```

After:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 002c756e 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0007 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PFLUSHR$+004           NOP
```

Note the change in the PCSR (task alias is 7).

PLOAD Instruction (Privileged)

Load ATC Entry

Syntax: PLOADR <fc>,<ea>
PLOADW <fc>,<ea>

Description

The PLOADR and PLOADW instructions simulate an access to the effective address, loading the appropriate descriptors into the Address Translation Cache (ATC) and updating the descriptors as necessary. The PLOADR instruction simulates a read access, while the PLOADW instruction simulates a write access. The function code argument specifies the function code bits FC3–FC0 for the simulated address. The function code can be immediate, in a data register, in the CPU SFC register, or in the CPU DFC register. Note that the 68851 requires 4 function code bits to accommodate DMA accesses. Since the SFC and DFC registers have only three bits, you cannot flush DMA addresses by specifying these registers as the function code. The 68030 has only 3 function code bits, and does not have this restriction.

If the Translation Control (TC) register E (enable) bit is not set, this instruction causes an MMU Illegal Operation exception through vector 57 (but only on a 68851).

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	-	-	-	-	-	-	-	-	-	-							•	•

B Not affected.
L Not affected. (68851 only.)
S Not affected.
A Not affected. (68851 only.)
W Not affected.
I Not affected.
M Not affected.
G Not affected. (68851 only.)
C Not affected. (68851 only.)
N Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn	000	Dn	
An	001	An	
(An)	010	An	•
(An)+	011	An	
-(An)	100	An	
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	
(d16,PC)	111	010	
(d8,PC,Xn)	111	011	
(bd,PC,Xn)	111	011	
([bd,PC,Xn],od)	111	011	
([bd,PC],Xn,od)	111	011	

The addressing mode specifies the address to be loaded, rather than a memory location containing the address. PLOAD is similar to the Load Effective Address (LEA) instruction in this regard. For instance, if the address whose descriptor is to be loaded is in a memory longword pointed to by A0, you must use the form PLOAD [(A0)], rather than PLOAD (A0). (The latter form would load the page whose address is in A0 rather than the longword to which A0 points.)

The effective address simply specifies the translation table entry to be loaded into the ATC. The PLOAD instruction simulates (but does not perform) an access to this address in order to properly update the ATC. The PLOADR instruction sets the U (Used) bit in the relevant descriptors. The PLOADW instruction sets both the U and W (Written) bits.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
0	0	1	0	0	0	R/W	0	0	0	0	FC				

The R/W bit is one to simulate a read access and zero to simulate a write. Bits 4–0 (FC) of the second instruction word identify the source of the function code value. These bits are interpreted as follows:

FC Field	Meaning
`00000`	Function code is in CPU SFC register
`00001`	Function code is in CPU DFC register
`1DDDD`	Function code is immediate, contained in bits 3–0 (DDDD)
`01RRR`	Function code is in Data register RRR

Example The PLOADR #1,(A0) simulates a user mode data read on the address contained in register A0. The machine code has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
0	0	1	0	0	0	R/W	0	0	0	0	FC				

1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	1

The hex value is $F010 2211. Executing the instruction yields:

Before:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 002c759e 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PLOAD$      PLOADR     #$1,(A0)
```

After:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 002c759e 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PLOAD$+004  NOP
```

PMOVE Instruction (Privileged)

MOVE MMU Register

Syntax:
PMOVE preg,<ea>
PMOVE <ea>,preg
(68030 Only) PMOVEFD <ea>,preg

Description

The PMOVE instruction transfers a single MMU register to or from an effective address operand. The length of the operation matches the register size: 1, 2, 4, or 8 bytes. The PMOVEFD instruction on the 68030 disables flushing the Address Translation Cache (ATC) when a new value is loaded into the CRP, SRP, TT0, TT1, or TC registers.

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	•	•	•	•	•	•	•	•	•	•		•	•	•	•		•	•

B	Set to source bit 15 if MMUSR is destination. Unchanged otherwise.
L	Set to source bit 14 if MMUSR is destination. Unchanged otherwise. (68851 only.)
S	Set to source bit 13 if MMUSR is destination. Unchanged otherwise.
A	Set to source bit 12 if MMUSR is destination. Unchanged otherwise. (68851 only.)
W	Set to source bit 11 if MMUSR is destination. Unchanged otherwise.
I	Set to source bit 10 if MMUSR is destination. Unchanged otherwise.
M	Set to source bit 09 if MMUSR is destination. Unchanged otherwise.
G	Set to source bit 08 if MMUSR is destination. Unchanged otherwise. (68851 only.)
C	Set to source bit 07 if MMUSR is destination. Unchanged otherwise. (68851 only.)
N	Set to source bits 2-0 if MMUSR is destination. Unchanged otherwise.

Valid Addressing Modes

Syntax	Mode	Reg	Source	Dest.	Syntax	Mode	Reg	Source	Dest.
Dn*	000	Dn	•	•	xxx.W	111	000	•	•
An*	001	An	•	•	xxx.L	111	001	•	•
(An)	010	An	•	•	#<data>	111	100	•	
(An)+	011	An	•	•					
-(An)	100	An	•	•					
(d16,An)	101	An	•	•	(d16,PC)	111	010	•	
(d8,An,Xn)	110	An	•	•	(d8,PC,Xn)	111	011	•	
(bd,An,Xn)	110	An	•	•	(bd,PC,Xn)	111	011	•	
([bd,An,Xn],od)	110	An	•	•	([bd,PC,Xn],od)	111	011	•	
([bd,An],Xn,od)	110	An	•	•	([bd,PC],Xn,od)	111	011	•	

☐ 68851 Only * -- CRP, DRP, or SRP not allowed as other operand

The 68851 supports several addressing modes that were not implemented on the 68030. (Note that the 68030 addressing modes are valid as both source and destination.) Also, the root pointer registers cannot be transferred to or from the CPU registers (8 bytes as opposed to 4).

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
0	MMU Register					R/W	FD	0	0	0	Break Register			0	0

The MMU Register field (bits 14–10 of the second instruction word) contains the ID of the affected MMU register. Bits in this field have the following interpretation:

MMU Register	MMU Register affected	Restrictions
00010	Transparent Translation Register 0 (TT0)	68030 Only
00011	Transparent Translation Register 1 (TT1)	68030 Only
10000	Translation Control Register (TC)	
10001	DMA Root Pointer (DRP)	68851 Only
10010	Supervisor Root Pointer (SRP)	
10011	CPU Root Pointer (CRP)	
10100	Current Access Level (CAL)	68851 Only
10101	Validate Access Level (VAL)	68851 Only
10110	Stack Change Control (SCC)	68851 Only
10111	Access Control (AC)	68851 Only
11000	MMU Status Register (MMUSR)	FD = 0
11001	PMMU Cache Status Register (PCSR)	68851 Only / Read only
11100	Breakpoint Acknowledge Data (BADx)	68851 Only / x = Break Register
11101	Breakpoint Acknowledge Control (BACx)	68851 Only / x = Break Register

The Read/Write bit (9) is 1 to transfer the MMU register to the effective address and zero to transfer the effective address to the MMU register. Note that the 68851 PMMU Cache Status Register (PCSR) cannot be modified with the PMOVE instruction (or any other instruction, for that matter). This register can only be transferred to the effective address, meaning that bit 9 must always be one when the MMU Register field has the value 11001.

Bit 8 of the second instruction word is the Flush Disable (FD) bit. This bit is valid only on the 68030, and only for registers other than the MMUSR. Setting this bit prevents the MMU from flushing the ATC when a Root pointer register is loaded. This bit must be zero on the 68851 and on the 68030 when the MMUSR is the target MMU register.

The Break Register field (Bits 4–2 of the second instruction word) identifies the breakpoint register BADx or BACx when the MMU Register field has the value 11100 or 11101. These bits must be zero for all other values of the MMU Register field. The Break Register field must always be zero on the 68030.

Example The PMOVE TC,D0 instruction copies the MMU Translation Control (TC) register into CPU register D0. The machine code has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
0	MMU Register					R/W	FD	0	0	0	Break Register			0	0

1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0

The hex value is $F000 4200. Executing the instruction yields:

Before:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PMOVE$      PMOVE      TC,D0
```

After:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 80f84500 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PMOVE$+004  NOP
```

PRESTORE Instruction (Privileged) Syntax: PRESTORE <ea>

Restore MMU State (68851 Only)

Description

The PRESTORE instruction loads the 68851 internal state from one of the state frames described in Figures 5-21 through 5-24. The CRP, SRP, MMUSR, CAL, VAL, and breakpoint registers (if enabled) are affected.

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	*	*	*	*	*	*	*	*	*	*							•	

B Loaded from restored state frame.
L Loaded from restored state frame. (68851 only).
S Loaded from restored state frame.
A Loaded from restored state frame. (68851 only).
W Loaded from restored state frame.
I Loaded from restored state frame.
M Loaded from restored state frame.
G Loaded from restored state frame. (68851 only).
C Loaded from restored state frame. (68851 only).
N Loaded from restored state frame.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn	000	Dn	
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	
(d16,PC)	111	010	•
(d8,PC,Xn)	111	011	•
(bd,PC,Xn)	111	011	•
([bd,PC,Xn],od)	111	011	•
([bd,PC],Xn,od)	111	011	•

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	1	Mode			Register		

Example The PRESTORE (A0) instruction loads the state frame whose address is contained in register A0. The machine code has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	1	Mode			Register		
1	1	1	1	0	0	0	1	0	1	0	1	0	0	0	0

The hex value is $F150. Executing the instruction yields:

Before:

```
USP=003b075a MSP=003b075a ISP =003b0758 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 003b1a68 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 003b0758
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00c064a7
TC =80f84500 PCSR = 0007 MMUSR=ffff BLSAWIMGCT--- Levels = 7
CAL=e0 VAL=e0 SCC=aa AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PRESTORE$   PRESTORE   (A0)
```

Register A0 points to an area with the following format:

```
03b1a68:      c0200000 0000ffee 40800050 7fff0202 . ......@..P ...
03b1a78:      40800050 7fff0202 ffff4200 010f0800 @..P .....B.....
03b1a88:      80f84500 00000000 00000000 00000000 ..E.............
```

This is an IDLE format state frame, as described in Figure 5-22.

After:

```
USP=003b075a MSP=003b075a ISP =003b0758 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 003b1a68 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 003b0758
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00c064a7
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=e0 VAL=e0 SCC=aa AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PRESTORE$+002         NOP
```

Note that the TC, CRP, SRP, CAL, VAL, and SCC registers were reloaded with their original contents.

PSAVE Instruction (Privileged)

Syntax: PSAVE <ea>

Save MMU State (68851 Only)

Description

The PSAVE instruction stores the 68851 internal state as one of the state frames described in Figures 5-21 through 5-24.

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	-	-	-	-	-	-	-	-	-	-							●	

B	Not affected.
L	Not affected. (68851 only.)
S	Not affected.
A	Not affected. (68851 only.)
W	Not affected.
I	Not affected.
M	Not affected.
G	Not affected. (68851 only.)
C	Not affected. (68851 only.)
N	Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn	000	Dn	
An	001	An	
(An)	010	An	●
(An)+	011	An	
-(An)	100	An	●
(d16,An)	101	An	●
(d8,An,Xn)	110	An	●
(bd,An,Xn)	110	An	●
([bd,An,Xn],od)	110	An	●
([bd,An],Xn,od)	110	An	●

Syntax	Mode	Reg	Valid
xxx.W	111	000	●
xxx.L	111	001	●
#<data>	111	100	
(d16,PC)	111	010	
(d8,PC,Xn)	111	011	
(bd,PC,Xn)	111	011	
([bd,PC,Xn],od)	111	011	
([bd,PC],Xn,od)	111	011	

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	Mode			Register		

Example The PSAVE (A0) instruction stores the state frame at the address contained in register A0. The machine code has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	Mode			Register		
1	1	1	1	0	0	0	1	0	0	0	1	0	0	0	0

The hex value is $F110 . Executing the instruction yields:

Before:

```
USP=002c6892 MSP=002c6892 ISP =002c6890 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 002c7a78 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c6890
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=aa AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PSAVE$      PSAVE      (A0)
```

Register A0 points to an area with the following format:

```
02c7a78:        00000000 00000001 00000002 00000003 ................
02c7a88:        00000004 00000005 00000006 00000007 ................
02c7a98:        00000008 00000009 0000000a 0000000b ................
```

After:

```
USP=002c6892 MSP=002c6892 ISP =002c6890 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 002c7a78 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c6890
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=aa AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PSAVE$+002  NOP
```

The memory area now has the following format:

```
02c7a78:      c0200000 0281ff00 40800050 7fff0202 . ......@..P ...
02c7a88:      40800050 7fff0202 0000000f 010f0800 @..P ...........
02c7a98:      80f84500 00000009 0000000a 0000000b ..E.............
```

This is an IDLE format state frame, as described in Figure 5-22.

PScc Instruction (Privileged)

Syntax: PScc <ea>

MMU Set Conditionally (68851 Only)

Description

The PScc sets a byte at the effective address to all ones if the specified MMU condition is true. The byte is set to zeros otherwise.

B Not affected.

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	-	-	-	-	-	-	-	-	-	-		•					•	

L Not affected. (68851 only.)
S Not affected.
A Not affected. (68851 only.)
W Not affected.
I Not affected.
M Not affected.
G Not affected. (68851 only.)
C Not affected. (68851 only.)
N Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn	000	Dn	•
An	001	An	
(An)	010	An	•
(An)+	011	An	•
-(An)	100	An	•
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	
(d16,PC)	111	010	
(d8,PC,Xn)	111	011	
(bd,PC,Xn)	111	011	
([bd,PC,Xn],od)	111	011	
([bd,PC],Xn,od)	111	011	

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	1	Mode			Register		
0	0	0	0	0	0	0	0	0	0	MMU Condition					

The MMU Condition allows you to check the state of a single MMUSR bit. Values for this field are

Condition	Mnemonic	Meaning
000000	BS -- B set	BUSERR indicator bit (15) in MMUSR is set.
000001	BC -- B clear	BUSERR indicator bit (15) in MMUSR is clear.
000010	LS -- L set	Lock indicator bit (14) in MMUSR is set.
000011	LC -- L clear	Lock indicator bit (14) in MMUSR is clear.
000100	SS -- S set	Supervisor indicator bit (13) in MMUSR is set.
000101	SC -- S clear	Supervisor indicator bit (13) in MMUSR is clear.
000110	AS -- A set	Access violation indicator bit (12) in MMUSR is set.
000111	AC -- A clear	Access violation indicator bit (12) in MMUSR is clear.
001000	WS -- W set	Write protection indicator bit (11) in MMUSR is set.
001001	WC -- W clear	Write protection indicator bit (11) in MMUSR is clear.
001010	IS -- I set	Invalid descriptor indicator bit (10) in MMUSR is set.
001101	IC -- I clear	Invalid descriptor indicator bit (10) in MMUSR is clear.
001110	GS -- G set	Gate indicator bit (9) in MMUSR is set.
001111	GC -- G clear	Gate indicator bit (9) in MMUSR is clear.

Example The PSBC D0 instruction sets the low-order byte of D0 to all ones if the MMUSR B bit (BUSERR) is clear. The machine code has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	1	Mode			Register		
0	0	0	0	0	0	0	0	0	0	MMU Condition					

1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

The hex value is $F040 0001. Executing the instruction yields:

Before:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PSCC$       PSBC       D0
```

After:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000ff 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PSCC$+004   NOP
```

PTEST Instruction (Privileged) Syntax:PTESTR <fc>,<ea>,#level[,An]
Test Logical Address PTESTW <fc>,<ea>,#level[,An]

Description

The PTESTR and PTESTW instructions set the condition codes in the MMUSR according to a translation table search using the specified effective address. The function code argument specifies the function code bits FC3–FC0 for the simulated address. The function code can be immediate, in a data register, in the CPU SFC register, or in the CPU DFC register. Note that the 68851 requires 4 function code bits to accommodate DMA accesses. Since the SFC and DFC registers have only 3 bits, you cannot test DMA addresses by specifying these registers as the function code. The 68030 has only 3 function code bits and does not have this restriction.

No descriptor bits are modified by this instruction. The physical address of the selected descriptor can optionally be placed in an address register. If the Root Pointer register selected by the function code contains a page descriptor (that is, there is no translation table for that root pointer), the address register receives a zero value.

PTEST is very useful in system software that uses the MMU. This instruction provides a way to validate pointers passed by a user program using the MMU hardware.

If the Translation Control (TC) register E (enable) bit is not set, this instruction causes an MMU Illegal Operation exception through vector 57 (but only on a 68851). A 68030 will generate an F-Line instruction on a PTEST with the level parameter equal to zero if an address register is specified.

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	*	*	*	*	*	*	*	*	*	*							•	•

B	Set if BUSERR occurred during translation table search. Cleared otherwise.
L	Set if a limit violation occurred. Cleared otherwise. (68851 only.)
S	Set if supervisor violation occurred. Cleared otherwise.
A	Set if access violation occurred. Cleared otherwise. (68851 only.)
W	Set if write protection violation occurred. Cleared otherwise.

I	Set if invalid descriptor encountered in table search. Cleared otherwise.
M	Set according to M bit of translation descriptor.
G	Set according to G bit of translation descriptor. (68851 only.)
C	Set according to C bit of translation descriptor. (68851 only.)
N	Set to number of table levels searched.

Two of these bits behave differently if the level parameter is zero. The B bit is set if the translation descriptor is not in the Address Translation Cache (ATC) and cleared otherwise. The N field is always set to zero whenever the level parameter is specified as zero. Specifying a non-zero level field forces the MMU to traverse the translation table, even if the desired entry is already in the ATC.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn	000	Dn	
An	001	An	
(An)	010	An	•
(An)+	011	An	
-(An)	100	An	
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	
(d16,PC)	111	010	
(d8,PC,Xn)	111	011	
(bd,PC,Xn)	111	011	
([bd,PC,Xn],od)	111	011	
([bd,PC],Xn,od)	111	011	

The addressing mode specifies the address to be tested, rather than a memory location containing the address. PTEST is similar to the Load Effective Address (LEA) instruction in this regard. For instance, if the address whose descriptor is to be tested is in a memory longword pointed to by A0, you must use the form PTEST [(A0)], rather than PTEST (A0). (The latter form would test the page whose address is in A0 rather than the longword to which A0 points.)

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
1	0	0	Level			R/W	A	Address Reg			FC				

The Level field (bits 12–10 of the second word of the instruction) contains the desired search level parameter. The R/W bit is one to test a read access and zero to test a write. If a write access is specified, the 68030 tests write protection violations; and the 68851 tests both write protection and access level violations. The A bit (8) is set to indicate the presence of an address register to receive the descriptor address. If this bit is set, the register number is in bits 7–5 of the second word.

Bits 4–0 (FC) of the second instruction word identify the source of the function code value. These bits are interpreted as follows:

FC Field	**Meaning**
`00000`	Function code is in CPU SFC register.
`00001`	Function code is in CPU DFC register.
`1DDDD`	Function code is immediate, contained in bits 3–0 (DDDD).
`01RRR`	Function code is in Data register RRR.

Example. The PTESTW #0,(A0),#$7,A1 sets the bits in the MMUSR according to the address in register A0. The function code is zero, and the number of allowed levels is at a maximum. The physical address of the descriptor is placed in register A1. The machine code has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
1	0	0	Level			R/W	A	Address Reg			FC				

1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0
1	0	0	1	1	1	0	1	0	0	1	1	0	0	0	0

The hex value is $F010 9D30. Executing the instruction yields:

Before:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 002c77d0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON  ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PTEST$      PTESTW     #$0,(A0),#$7,A1
```

After:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 002c77d0 40800058 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0281 ------M-C---- Levels = 1
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PTEST$+004  NOP
```

PTRAPcc Instruction (Privileged)

Trap on MMU Condition (68851 Only)

Syntax: PTRAPcc
PTRAPcc.W #data
PTRAPcc.L #data

Description

The PTRAPcc instruction allows you to trap on a condition evaluated by the MMU. This instruction is similar to the FTRAPcc for floating point and TRAPcc on the main processor. You can optionally follow the instruction with 16 or 32 bits of immediate data. The Macintosh MPW system uses the mnemonic PTPcc rather than PTRAPcc.

Exception vector 7 is used for PTRAPcc, FTRAPcc, and TRAPcc. These exceptions all use a format 2 stack frame, as shown in Figure 2-39. Note that offsets $8 and $A of this stack frame point to the first instruction word. This is helpful in retrieving the immediate data following the trap. The CPU adjusts the PC (offsets $2 and $4) so that an RTE will skip the immediate data.

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	-	-	-	-	-	-	-	-	-	-			•	•			•	

B Not affected.
L Not affected. (68851 only.)
S Not affected.
A Not affected. (68851 only.)
W Not affected.
I Not affected.
M Not affected.
G Not affected. (68851 only.)
C Not affected. (68851 only.)
N Not affected.

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	1	1	1	1	Mode		
0	0	0	0	0	0	0	0	0	0	MMU Condition					
16-bit immediate data if Mode = 010															
32bit immediate data if Mode = 011															

The Mode field (bits 2–0 of the first word) controls how much immediate data follows the instruction word. This field is 100 for no immediate data, 010 for 16 bits of immediate data, and 011 for 32 bits of immediate data.

The MMU Condition allows you to check the state of a single MMUSR bit. Values for this field are

Condition	Mnemonic	Meaning
000000	BS -- B set	BUSERR indicator bit (15) in MMUSR is set.
000001	BC -- B clear	BUSERR indicator bit (15) in MMUSR is clear.
000010	LS -- L set	Lock indicator bit (14) in MMUSR is set.
000011	LC -- L clear	Lock indicator bit (14) in MMUSR is clear.
000100	SS -- S set	Supervisor indicator bit (13) in MMUSR is set.
000101	SC -- S clear	Supervisor indicator bit (13) in MMUSR is clear.
000110	AS -- A set	Access violation indicator bit (12) in MMUSR is set.
000111	AC -- A clear	Access violation indicator bit (12) in MMUSR is clear.
001000	WS -- W set	Write protection indicator bit (11) in MMUSR is set.
001001	WC -- W clear	Write protection indicator bit (11) in MMUSR is clear.
001010	IS -- I set	Invalid descriptor indicator bit (10) in MMUSR is set.
001101	IC -- I clear	Invalid descriptor indicator bit (10) in MMUSR is clear.
001110	GS -- G set	Gate indicator bit (9) in MMUSR is set.
001111	GC -- G clear	Gate indicator bit (9) in MMUSR is clear.

Example The TRAPBS #$89ABCDEF instruction traps if the MMUSR B (BUSERR) condition code is set. The machine code has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	1	1	1	1	Mode		
0	0	0	0	0	0	0	0	0	0	MMU Condition					
16-bit immediate data if Mode = 010															
32bit immediate data if Mode = 011															

1	1	1	1	0	0	0	0	0	1	1	1	1	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1	1	0	1	0	1	0	1	1
1	1	0	0	1	1	0	1	1	1	0	1	1	1	1	1

The hex value is $F07B 0000 89AB CDEF. Executing the instruction yields:

Before:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON  ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PTRAPCC$    PTRAPBS.L  #$89ABCDEF
```

After:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000 MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON  ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE Table @ 40800050
PC =PTRAPCC$+008 NOP
```

Since the B bit was clear, the trap was not taken. Note that the PC value following the instruction shows the instruction length.

PVALID Instruction (Privileged)

Validate Address (68851 Only)

Syntax: PVALID VAL,<ea>
PVALID An,<ea>

Description

The PVALID instruction provides a way of validating pointers within the 68851 access level control mechanism. The high bits of an address are compared either to those in an address register or those in the Validate Access Level (VAL) register. If the access is invalid, an MMU Access Level exception through vector 58 takes place. Otherwise, execution continues with the next instruction. PVALID always causes an Access Level exception if the Module Control (MC) bit of the Access Control (AC) register is not set.

MMUSR	B	L	S	A	W	I	M	G	C	N	Sizes	Byte	Word	Long	Quad	MMUs	851	030
	-	-	-	-	-	-	-	-	-	-				•			•	

B Not affected.
L Not affected. (68851 only.)
S Not affected.
A Not affected. (68851 only.)
W Not affected.
I Not affected.
M Not affected.
G Not affected. (68851 only.)
C Not affected. (68851 only.)
N Not affected.

Valid Addressing Modes

Syntax	Mode	Reg	Valid
Dn	000	Dn	
An	001	An	
(An)	010	An	•
(An)+	011	An	
-(An)	100	An	
(d16,An)	101	An	•
(d8,An,Xn)	110	An	•
(bd,An,Xn)	110	An	•
([bd,An,Xn],od)	110	An	•
([bd,An],Xn,od)	110	An	•

Syntax	Mode	Reg	Valid
xxx.W	111	000	•
xxx.L	111	001	•
#<data>	111	100	
(d16,PC)	111	010	
(d8,PC,Xn)	111	011	
(bd,PC,Xn)	111	011	
([bd,PC,Xn],od)	111	011	
([bd,PC],Xn,od)	111	011	

The addressing mode specifies the address to be validated, rather than a memory location containing the address. PVALID is similar to the Load Effective Address (LEA) instruction in this regard. For instance, if the address whose descriptor is to be validated is in a memory longword pointed to by A0, you must use the form PVALID [(A0)], rather than PVALID (A0). (The latter form would validate the address in A0 rather than the address in the longword to which A0 points.)

Instruction Format

VAL Register contains Access Level:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0

Address Register contains Access Level:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
0	0	1	0	1	1	0	0	0	0	0	0	0	Address Reg		

The address register number is contained in bits 2–0 of the second word of the instruction.

Example The PVALID VAL,(A0) validates the address in register A0 against the Validate Access Level (VAL) register. The machine code has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	Mode			Register		
0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0

1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0

The hex value is $F010 2800. Executing the instruction yields:

Before:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000  MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON  ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE  Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE  Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE  Table @ 40800050
PC =PVALID$     PVALID     VAL,(A0)
```

After:

```
USP=002c64e6 MSP=002c64e6 ISP =002c64e4 SR =2700 -- -- SI PL=7 -----
D0: 000000d0 000000d1 000000d2 000000d3 000000d4 000000d5 000000d6 000000d7
A0: 000000a0 000000a1 000000a2 000000a3 000000a4 000000a5 000000a6 002c64e4
SFC=00000001 DFC=00000001 CACR=00000001 VBR=00000000 CAAR=00133b56
TC =80f84500 PCSR = 0000  MMUSR=0000 ------------- Levels = 0
CAL=00 VAL=00 SCC=fe AC=0083 (MC=ON  ALC=Disabled MDS=64-byte)
CRP=7fff0202 40800050 U V 4-BYTE  Table @ 40800050
SRP=7fff0202 40800050 U V 4-BYTE  Table @ 40800050
DRP=7fff0202 40800050 U V 4-BYTE  Table @ 40800050
PC =PVALID$+004            NOP
```

Since the high bits of A0 match the bits in the VAL register, the address is valid, and no trap occurs.

CHAPTER

6

Program Examples

"Example is not the main thing in influencing others.
It is the only thing." — *Albert Schweitzer*

Introduction

This chapter explores some of the fine points of programming in assembly language on 68000 chips. Statements are often made predicting the demise of this type of programming. Claims are made that high-level language compilers can achieve results rivaling those of a good assembly language programmer. There is some merit to this argument because a compiler can, in fact, produce code that rivals the best hand-coding for large sections of code. Furthermore, as Brooks notes in *The Mythical Man-Month* (Addison-Wesley, 1975), you can write software up to five times faster in a high-level language than in assembly language.

Although compilers have many strengths, there are also arguments that favor assembly language. While it is quite likely that no new major software project will be done completely in assembly language, it is also quite likely that no major project will be done completely without assembly language. There are two primary reasons for this situation: efficiency and functionality. When considering efficiency, remember the 80/20 rule that applies to software: 80 per cent of the time is spent in 20 per cent of the code. It is also true that you can hand-code small sections of software that run much faster than the same code produced by a compiler. Thus, the 80/20 rule means you can produce substantial performance improvements by hand-coding a relatively small amount of software to assembly language. When considering functionality, remember that some functions are available only to assembly language software. Few compilers allow you to directly execute specialized instructions, such as those dealing with the MMU, interrupts, or special-purpose registers like the 68020/30 Master Stack Pointer (MSP) and Cache Control Register (CACR). Thus, you must resort to assembly language to access these and similar hardware features.

Most software today consists of a marriage of high-level language code to a small amount of assembly language. This chapter is oriented toward this type of software architecture. The next section of this chapter describes how to interface assembly language and a high-level language. We have chosen C (and by extension C++) as the vehicle for the program examples. Every attempt has been made to make these programs usable on as many systems as possible. In the event of a conflict, we have chosen the Macintosh Programmer's Workshop (MPW) standard for the Apple Macintosh computer.

Interfacing to a High-level Language

Binding a high-level language to assembly language involves three principal components: data types, the procedure calling sequence, and accessing global data. We will explore three development environments: the Apple MPW C compiler, the Apple MPW Pascal Compiler, and the UNIX System V pcc compiler.

Each of the three components (data structures, procedure calling, and global data) for each of the three compilers (Apple MPW C, Apple MPW Pascal, and UNIX pcc) is explained below. The discussions are grouped by compiler. The components are explained three times, once for each compiler.

Apple MPW C

Version 3.0 of the Apple Macintosh Programmer's Workshop (MPW) uses a new C compiler, which was written within Apple rather than purchased from outside. The previous release of MPW used a compiler that was purchased from Green Hills Software, Inc. With one minor exception, the two compilers are compatible.

C on the Macintosh also has one very interesting quirk: The compiler must be able to generate Pascal format procedure calls because they are required by the Macintosh operating system. The section below on Macintosh Pascal describes this calling sequence.

You must specify the CASE ON directive to the assembler to force lower- or mixed-case identifiers to match those generated by the C compiler. Without this directive, the name foo is seen by the linker as FOO.

Data Types C supports several integer and floating point data types that are (almost) standard across compilers and processors. Incompatibilities arise when comparing the memory size and precision of these standard data types. For instance, an integer variable may occupy 16 bits on one compiler, but 32 bits on another. These data types and their sizes are listed below:

Data Type	Size (bits)	Notes
char	8	1-byte signed or unsigned integer
short	16	2-byte signed or unsigned integer
int	32	4-byte signed or unsigned integer
long	32	4-byte signed or unsigned integer
float	32	68881/2 Single precision data type
double	64	68881/2 Double precision data type
extended	96 (80)	(non-standard) 68881/2 extended data type
(pointers)	32	Map directly into 68000 addresses

C compilers are the most compatible when using integer arithmetic. The major difference between 68000 C compilers is whether the 'int' data type is 16 bits (Atari TOS system for the ST series, Lightspeed C on the Mac, and CP/M-68K) or 32 bits (practically everyone else). The 16-bit integers are notably faster on the 68000 and 68010 CPUs. If 32-bit integers are aligned on longword boundaries, there is very little speed difference on a 68020 or 68030 with a 32-bit data bus. The 16-bit integers also save memory space. This choice has ramifications for the procedure calling standard, as we shall soon see.

Pointers to any of these data types occupy 32 bits. Some old UNIX code assumed that pointers and integers were the same length. MPW C is compatible with code written this way.

Floating point is another matter. The original C specification for UNIX treated floating point almost as an afterthought. Although the C standard has both single- and double-precision floating point, many C implementations use only double precision. The extended variable type in MPW C allows direct manipulation of this 68881/2 data type from the high-level language. You can specify either extended or long double as the type specifier in your program. The float, double, and extended C data types map directly onto the 68881/2 single, double, and extended precision data types. MPW C does not map single precision into double precision, as do many older UNIX compilers.

Structures are another area where incompatibility may creep in. Some 68000 compilers align structure members to longword boundaries, while others do not. For instance, the structure

```
struct  foo     {       /* Misaligned data sample       */
        short   a;      /* 16 bits                      */
        long    b;      /* 32 bits                      */
        };              /*                              */
```

represents 6 bytes on MPW and some other compilers, but 8 bytes on other compilers. On these latter compilers, 2 bytes of padding are inserted following the 16-bit quantity a, so that 32-bit data types are aligned on longword boundaries. Although this made no difference on the 68000 and 68010, accessing a misaligned longword requires two bus cycles instead of one on a 68020 or 68030. Since many of the early Macintosh data structures have misaligned longwords, the MPW C designers were forced to use misaligned structures.

Strings in C are arrays of consecutive bytes, with a null byte at the end. The Macintosh designers picked the Pascal standard for the Macintosh O/S, meaning that C strings must be converted for use with Macintosh system calls. Both Pascal and C string formats are shown in Figure 6-3 below. You can cause the C compiler to generate a Pascal format string using the \P escape, as described in the section on the Pascal compiler.

Another deviation from typical practice is that the value of the newline character (\n) is that for ASCII carriage return ($0D) rather than line feed ($0A). The newline and return (\r) characters have the same value in MPW C, which can cause code ported from another environment to malfunction. This feature is contrary to the way C works in most other environments.

Procedure Calling Standard C has functions, notably printf and scanf, which take a variable number of arguments. Handling this requirement led to the calling sequence used by most C compilers, including MPW C.

Parameters are pushed onto the stack in reverse order, leaving the first parameter at the lowest address. The called procedure is invoked with a JSR or BSR instruction, leaving the return address at offset zero from the stack pointer, with the first parameter at offset 4. For instance, the C call:

```
long  a,b,c,r,cfunc();

r = cfunc(a,b,c);
```

generates the following assembly language:

```
MOVE.L  c,-(SP)          ;   Parameters in
MOVE.L  b,-(SP)          ;                reverse
MOVE.L  a,-(SP)          ;                        order
JSR     cfunc            ;   Call the function
LEA     12(SP),SP        ;   Pop parameters
MOVE.L  D0,r             ;   Do something with return value
```

Note that the caller removes the arguments from the stack following the call and that the return value comes back in D0. FP0 is used for floating point return values (which must be in extended format due to the 68881). You need not preserve the values in registers D0–D1, A0–A1, and FP0 (and C functions which you call from assembly language will not preserve them, either).

The interesting part comes when the items passed into and returned from the called procedure are not long variable types. Ancient UNIX systems were based on the PDP-11, where the char, int, and pointer data types were all passed in as the same length (16 bits). The float and double data types were both passed as double (64 bits). Arrays and structures were passed by address. Note that this scheme meant that there were only two possible data types coming into a function. If you didn't use floating point (and most programs don't), there was only one.

The introduction of the long (32 bit) data type, and the move to 32-bit processors moved the standard away from 16 bits to 32. UNIX compilers on the 68000 pass char, int, long, and pointer data types as 32 bits, and float and double as double (64 bits). With the exception of the new feature of passing structures by value, there are still only two possible data types, and the old UNIX code (which assumed this arrangement) still functions.

MPW is compatible with this standard for the most part. All integer and pointer data types are passed as 32 bits. Floating point, however, is passed as extended (80/96 bit) precision rather than double precision. Inside a procedure, however, each data type occupies only as much memory as required. When extending signed integer data types, the sign bit is extended into the upper bits. For unsigned data types, the upper bits are zeroed, as shown in Figure 6-1.

The x characters indicate data bits, and the s characters indicate the sign bit. Floating point expansion is performed with FMOVE instructions, so that all conversion is done by the 68881/2 chip (or its software equivalent).

Structures are also very interesting. Passing a structure to a function creates a copy of the structure on the stack. It is extremely easy to pass the entire structure, when what you really wanted was the address of the structure. MPW C 2.0 used a comparatively slow byte-by-byte copy process to generate the stack structure. The MPW C 3.0 technique is far more efficient, but still awkward.

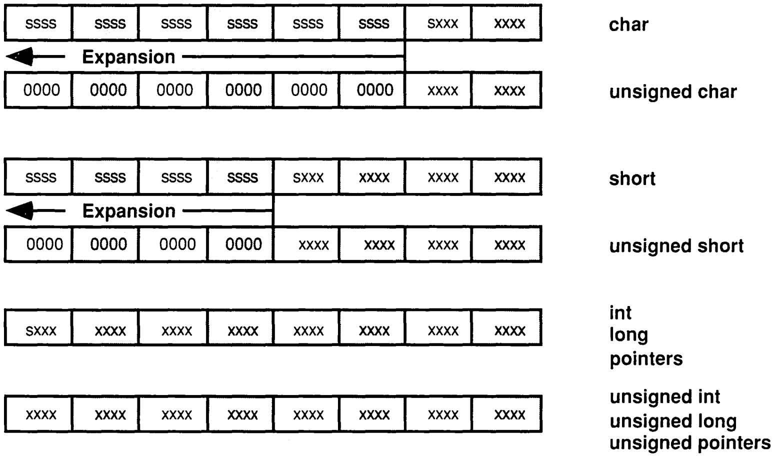

ssss ssss ssss ssss ssss ssss sxxx xxxx
Expansion
0000 0000 0000 0000 0000 0000 xxxx xxxx
char
unsigned char
ssss ssss ssss ssss sxxx xxxx xxxx xxxx
Expansion
0000 0000 0000 0000 xxxx xxxx xxxx xxxx
short
unsigned short
sxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
int
long
pointers
xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
unsigned int
unsigned long
unsigned pointers

Figure 6-1. Integer Data Type Expansion

Quantities passed back in D0 as function return values are treated differently between the two compilers. On MPW 3.0, when you pass back a short or char value (less than 32 bits), the upper bits of the register are garbage. This saves an instruction in the procedure exit code sequence. MPW 2.0 always extends these values, as shown in Figure 6-1.

On the 68020 and 68030 chips, it is advantageous to keep the stack longword-aligned. Since stack accesses are predominantly 32 bits, keeping the stack longword reduces the number of memory accesses required to access a data item. MPW C, unfortunately, makes no effort to perform this optimization.

The final topic associated with the procedure calling standard is that of a *stack frame*. You have seen how the arguments are pushed onto the stack. In MPW C, the called routine usually issues a LINK instruction using A6 as the frame pointer. The offset parameter in the LINK is the size of the local variable area required by the procedure. If all procedures follow this standard, it is possible to trace procedure calls all the way back through a program. This situation is shown in Figure 6-2.

By sacrificing the use of an address register (A6), you no longer have to worry about how much data has been pushed on and popped off the stack since the procedure started. This is especially valuable in a high-level language where a function call can use the return value of another function call as a parameter. Using the stack frame technique, local variables are at fixed negative offsets from A6 and parameters are at positive offsets from A6. You can

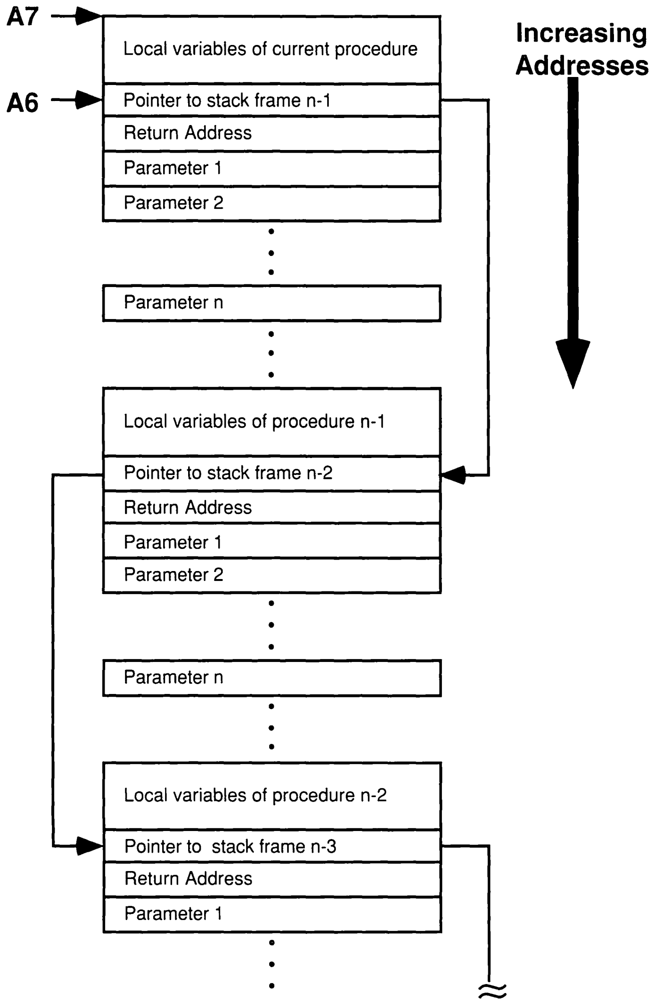

A7
A6
Local variables of current procedure
Pointer to stack frame n-1
Return Address
Parameter 1
Parameter 2
Parameter n
Local variables of procedure n-1
Pointer to stack frame n-2
Return Address
Parameter 1
Parameter 2
Parameter n
Local variables of procedure n-2
Pointer to stack frame n-3
Return Address
Parameter 1
Increasing Addresses

Figure 6-2. Stack Frame Structure

change A7 without affecting the offsets, which makes referencing stack-based variables much easier.

As mentioned previously, a second benefit is the ability to trace back the calling sequence (including parameters) from any function. Most 68000 debuggers support this feature. On the Macintosh MACSbug, the command is known as sc (Stack Crawl). The UNIX (A/UX) adb debugger c command performs the same function.

The major difference between MPW C version 3 and MPW C version 2 is that the older version always used a 32-bit return value from a function. The 16- or 8-bit return values were sign or zero extended (for signed or unsigned data types). The newer compiler reserves the right to leave garbage in the upper bits of short or char return values.

Accessing Global Data MPW C runs on a Macintosh under the Macintosh O/S. One of the requirements of this system is that programs cannot use the 68000 absolute addressing modes for accessing data items inside the program itself. These addressing modes can be used for low memory (addresses 0–32K), however.

The Macintosh standard decrees that you must use register A5 as a global frame of reference for items within your program. Negative offsets from A5 are used for global variables and positive offsets are used for parameters passed from the system and for a table of routine addresses that cross Macintosh segment boundaries. This information is discussed in *Inside Macintosh, Volume II* (Addison-Wesley, 1985), in the The Segment Loader section.

There are two practical implications of the use of A5: you can't modify A5 without saving it and restoring it, and all references to global data must use A5 and the address register with displacement addressing mode. Fortunately, MPW performs this latter action automatically. For instance, suppose the global variable "foo" is declared in C as:

```
int   foo;              /* 32 bits (signed) of data     */
```

(outside a procedure definition). The 4 bytes comprising foo are thus in the A5 global data area. To reference this variable from assembly language, you must use an IMPORT statement either within a DATA area, or with a DATA qualifier. This latter technique is

```
IMPORT   foo:DATA          ;    Tell MPW where it is.
MOVE.L   foo,D0            ;    Fetch the value.
```

The assembler will also accept references that explicitly reference A5, for example, foo(A5) in the code section above. Items that are IMPORTed can be either CODE or DATA. The default is the type of segment where the IMPORT directive occurs. For instance, in a CODE segment, the statements

```
CODE                       ;    Explicitly code
IMPORT   foo               ;    Incorrect IMPORT for data
MOVE.L   foo,D0            ;    Generates a bogus reference
```

would cause the linker to complain about references to the wrong segment (given the same C declaration above). If, however, you specify

```
DATA                    ;   References to data area now
IMPORT   foo            ;   Correct IMPORT for data
CODE                    ;   Now generate code
MOVE.L   foo,D0         ;   Generates a correct reference
```

the program works correctly.

One side effect is that many languages on the Macintosh restrict you to 32K of global data (the limit imposed by the address register with displacement addressing mode). MPW 3.0 C falls into this category.

To declare data in assembly language that is to be referenced from C, you must first specify a DATA directive (indicating that subsequent directives apply to data). You can then use the DC (define constant) or DS (define storage) to declare initialized and uninitialized data areas. For instance, consider the following two code segments.

C:

```
extern      int      foo;        /* Reference assembler variable    */
main()
{
            printf("Foo is: %d\n",foo);/* Print the value            */
}
```

Assembler:

```
            DATA                ;   Now in data segment
            EXPORT   foo        ;   foo to be referenced in C
foo:        DC.L     1234       ;   define
```

The C program would print “Foo is: 1234” on execution.

Referencing data declared in a CODE segment from C is not allowed. Note that the C compiler can generate the data for strings within the CODE segment containing the strings. This is a useful technique for circumventing the 32K restriction on global data.

UNIX pcc Compiler

The UNIX pcc (Portable C Compiler) translator is a derivative of the original work of the same name for the PDP-11. For many years, this compiler defined C for implementors who wrote C compilers for non-UNIX systems. The MPW C compiler is quite compatible with this older implementation, differing mainly in the area of floating point.

Data Types pcc implements all of the data types supported by MPW, with the exception of the extended or double long floating point data type. Floating point computations in pcc are limited to single and double precision for float and double data types, respectively. Otherwise, pcc is compatible with MPW C.

Procedure Calling Standard pcc and MPW C also differ slightly in the procedure calling standard. The order and size of integer parameters on the stack is identical. Floating point values, however, are passed in double-precision format rather than extended-precision format (64 bits versus 96). Return values are passed in register D0, and the full 32-bit value is valid (sign or zero extended, as above). Pointer values are passed back in both A0 and D0, however, for compatibility with older Motorola-supplied software tools. Floating point values are returned in register FP0.

When structures are passed by value, the code generated is comparable to MPW C 3.0. The compiler takes the extra step of longword alignment for the stack pointer, thereby assuring faster access to 32-bit stack variables.

When structures are returned by value, a structure of 32 bits or less is passed back in D0. If the structure is greater than 32 bits, the address of that structure is passed back in D0 and A0. The caller is expected to perform the copy into its local storage. Note that this technique is not reentrant.

The stack frame used by pcc is essentially the same as that used by MPW C 3.0. The compiler places an extra 32-bit word on the stack, which is used as the first parameter pushed when making a subroutine call. Note that this extra longword is apparent only for the current procedure and for procedures called with no parameters. Debugging tracebacks are the same as with the MPW compilers.

Accessing Global Data UNIX is consistent with many other systems in that absolute addressing is the method of choice for accessing global data. It is possible with UNIX, for instance, to declare a data item whose value is the address of a code or data item. This is far more difficult on the Macintosh.

UNIX assembly language global data items are imported or exported with the global directive. This directive also works for text (code) labels. If the label is defined in the current assembly, the global directive is equivalent to an EXPORT. If the label is not defined, global works like IMPORT.

The UNIX tools allow you to define data in the text (CODE) segment. However, you cannot modify this data on many versions of UNIX, since some versions of UNIX write-protect the code area.

Macintosh Pascal

Macintosh Pascal was intended to be the language of choice for writing Macintosh software. The Macintosh O/S requires this calling sequence for most system calls. Most Macintosh compilers (like the MPW C described above) can generate this type of calling sequence if it is not the default for a particular compiler. The Pascal calling sequence differs from the C conventions in two ways: parameters are pushed onto the stack in the opposite order and function values are returned on the stack instead of a register.

Data Types Pascal data types are also different. The Pascal data types are as follows:

Data Type	Size (bits)	Notes
BOOLEAN	8	1-byte which is either 0 or non-zero
SIGNEDBYTE	8	1-byte signed integer
BYTE	16	0..255 in low-order byte
CHAR	16	Extended ASCII in low-order byte
STRING[n]	(n+1)*8	n+1 byte array of 8-bit characters
STR255	2048(max)	1-255 byte character string
INTEGER	16	2-byte signed integer
UNSIGNEDINT	16	2-byte unsigned integer
LONGINTEGER	32	4-byte signed or unsigned integer
REAL	32	68881/2 Single precision data type
DOUBLE	64	68881/2 Double precision data type
EXTENDED	96 (80)	(non-standard) 68881/2 extended data type
PTR	32	Address of data
HANDLE	32	Address of Macintosh Master Pointer
(pointers)	32	type^ or type@ Pascal-style addresses

There are two principal areas of difference between C and Pascal. Where C integer types are all passed to procedures as 32-bit quantities, Pascal can pass either 16 or 32 bits, as explained in the section below.

String variables are the other area of difference. C strings, as noted above, consist of an array of consecutive bytes, terminated by a null byte. Conversely, Pascal strings have a count byte followed by the data bytes as illustrated in Figure 6-3.

Note that Pascal normally numbers characters from 1, while C numbers them from zero (in the illustration above, m = n − 1). An advantage of the Pascal scheme is that you need not traverse the string to find out how long it is; on the other hand, the string cannot exceed 255 (data) bytes. The STR255 data type is precisely this string format and is used by the Macintosh system software whenever the name of something is required.

MPW C can generate a Pascal string using the '\P' escape character as the first character in a quoted string (the P must be capitalized). The compiler places a count at the front and a null byte at the end. For example, the string "\PHello" generates a byte with the value 5, the five characters 'H', 'e', 'l', 'l', 'o', and a null (zero) byte, for a total of 7 bytes. This is useful for invoking Macintosh system calls or Pascal subroutines from C.

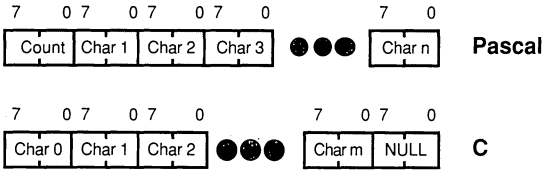


Figure 6-3. String Formats

Procedure Calling Standard Pascal FUNCTIONs return values on the stack. Parameters are pushed in left to right order, which is the opposite of the ordering used by the C compiler. Taking the same example we used above, the calling sequence (translated to Pascal) is

```
VAR
  a,b,c,r              :     LONGINT;

FUNCTION cfunc (a,b,c : LONGINT) : LONGINT; EXTERNAL;

BEGIN
  r := cfunc(a,b,c);
END;
```

This sequence generates the following assembly language:

```
LEA       -4(SP),SP      ;      Room for return value
MOVE.L    a,-(SP)        ;      Reverse
MOVE.L    b,-(SP)        ;              order
MOVE.L    c,-(SP)        ;                    from C
JSR       cfunc          ;      Call the function
MOVE.L    (SP)+,r        ;      Set return value
```

As with C, there are complications. Any VAR parameter is passed by address, not by value. Any non-VAR parameter over 4 bytes is passed by address, rather than by value. This includes double and extended floating point data types (but not real). Records (structures) are also passed by address, unless the record fits in 32 bits (4 bytes), in which case it is passed by value.

The same is true for return values. The parameter is passed back by value if it is 4 bytes or less. Otherwise, the caller places the address of the return area on the stack, and the called procedure stores the return value at that address.

The 8-bit parameters are extended to 16 bits, but 16-bit parameters are not extended to 32 bits. Thus, you have only two possible parameter lengths in the stack frame: 16 and 32 bits. Variable parameter lists are not possible.

Register conventions are slightly different from MPW C. You need not preserve the values in registers D0–D2, A0–A1, and FP0–FP3. (Pascal functions (including Macintosh ROM calls) that you call from assembly language will not preserve them either).

The A6 convention for the stack frame is the same. Stack frames for C and Pascal differ only in the use of the stack for function return values and the parameter conventions described earlier. Negative offsets from A6 are used in both languages for local variables, and positive offsets for parameters. The global data area (the VAR area associated with the main Pascal procedure) uses the Macintosh standard A5 area for global data.

Accessing Global Data Accessing global data is not possible. Using Pascal and assembly language together requires that data be enveloped in a procedure or function in the file in which it is declared.

Performance Improvements

This section describes some general guidelines for writing assembly code that performs well on 68000 family CPUs. We will pay special attention to the 68020 and 68030 in this section since these chips will dominate this family within a few years.

In general, to optimize a program, you should focus on the 20 percent of the code that takes 80 per cent of the time. Within this section, you may find that the performance is inhibited by a handful of routines. By recoding these routines in assembler, you may see a dramatic improvement in the performance of the entire program by optimizing a small part. See the section on the Dhrystone benchmark later in the chapter for an example.

Here are several very general suggestions for making programs run faster on 68000 family processors.

1. Minimize the total number of times the processor goes to memory. Most optimizations have this at their root. Placing data in registers more often, minimizing the number of instructions in a loop, using non-standard calling sequences, and many other optimizations are an attempt to reduce the processor-to-memory bottleneck. (The term 'Von Neumann bottleneck' refers to the processor-to-memory interface).
2. Place frequently-accessed data in registers. When you access data in a register, you do not access memory and the instruction is smaller and faster.
3. Reduce the total number of instructions required to perform a given function. In a loop, you should perform several iterations in the body of the loop rather than just one. This technique, called loop unrolling, reduces the number of times the loop overhead is executed and yields higher performance.
4. When transferring several items between memory and registers, use the MOVEM instructions if possible, but be careful about which registers you assign to which data items. Note that this may not be the best technique on the 68020 and 68030 processors (which have an instruction cache).
5. Sometimes you can use larger transactions from memory to reduce execution time. See the character string examples that use longword accesses for character strings (which are basically 8-bit data instead of 32-bit).

6. For programs that use the 68881 or 68882 floating point unit, attempt to interleave floating point instructions with integer instructions to take advantage of parallelism. On a 68882, attempt to maximize concurrency using the principles outlined in Chapter 4.

Another thing to consider in writing any software is to choose the best algorithm for the job. No amount of performance optimization will save a poor algorithm. This is true regardless of what language you use. For example, even the best linear search implementation will not perform as well as a mediocre binary search implementation.

Programming Examples

The remainder of this chapter consists of programming examples that proceed from the simple to the more complex. In all cases, the source code printed here will run under the Macintosh MPW version 3.0 development system.

C String Functions

As a simple example, we will consider two C language string support routines, strcpy() and strcmp(), which copy and compare two null-terminated strings. The Dhrystone benchmark, which is often used to compare computers for integer CPU performance, is overly dependent on these two string comparison routines. We will show how performance can be improved with just a few lines of assembly language code.

strcpy( ) One of the most common routines in C is the one that copies a null-terminated string to another memory area. This function is called strcpy and has the following calling sequence:

```
char   *src,*dest,*ret;           /* Source, destination, return  */
char   *strcpy();                 /* Returns pointer to dest      */

ret = strcpy(dest,src);           /* Note right to left syntax    */
```

A simple C version of this routine is

```
char   *strcpy(dest,src)
char   *dest;                 /* -> destination area,        */
char   *src;                  /* -> Source area              */
{                             /*---------------------------- */
       register char *p = dest; /* Temporary                 */
       while(*p++ = *src++)   /* Do the copy                 */
            ;                 /* while() does it all.        */
       return(dest);          /* Return source pointer       */
}                             /* Simple!                     */
```

The requirement for returning the original value of the destination is a bit unwieldy. The MPW 3.0 C compiler generates the following code for this function:

```
0000: 4E56 0000 strcpy:  LINK     A6,#$0000
0004: 48E7 0018          MOVEM.L  A3/A4,-(A7)
0008: 286E 000C          MOVEA.L  $000C(A6),A4
000C: 266E 0008          MOVEA.L  $0008(A6),A3
0010: 16DC               MOVE.B   (A4)+,(A3)+
0012: 66FC               BNE.S    *-$0002         ; 00000010
0014: 202E 0008          MOVE.L   $0008(A6),D0
0018: 4CEE 1800 FFF8     MOVEM.L  -$0008(A6),A3/A4
001E: 4E5E               UNLK     A6
0020: 4E75               RTS
```

(This listing was derived from the output of the MPW dumpobj utility.) Notice the compiler used the LINK/UNLK mechanism, even though strcpy() calls no subroutines. Note also that it modifies D0 and two address registers. Since the calling standard for MPW C allows us to modify D0, A0, and A1 without saving them, a hand-recoding of strcpy() yields this optimization:

```
strcpy: MOVEM.L     4(SP),A0/A1     ;    MPW Doesn't save these
        MOVE.L      A0,D0           ;    Return Value
loop:   MOVE.B      (A1)+,(A0)+     ;    Copy a byte
        BNE.S       loop            ;    Loop
        RTS                         ;    Done
```

There are three overhead instructions for this function (MOVEM, MOVE.L, and RTS), as opposed to eight for the compiler-generated output. You should expect the optimized version to perform well for short strings and for the improvement to all but vanish for long strings (the MOVE.B/BNE.S loop will dominate the execution time). Indeed, the execution statistics on a Macintosh II are

```
Bytes = 0001 times: 013.83 / 008.17 (40%)
Bytes = 0002 times: 014.67 / 009.00 (38%)
Bytes = 0004 times: 016.50 / 010.83 (34%)
Bytes = 0008 times: 020.33 / 014.50 (28%)
Bytes = 0016 times: 027.50 / 021.67 (21%)
Bytes = 0032 times: 042.00 / 036.17 (13%)
Bytes = 0064 times: 071.16 / 065.50 (07%)
Bytes = 0128 times: 129.50 / 124.17 (04%)
Bytes = 0256 times: 246.33 / 240.50 (02%)
Bytes = 0512 times: 479.17 / 475.50 (00%)
```

Times are in microseconds and the percentages are the percent improvement of assembly language over C. As expected, the improvement quickly vanishes as strings get larger than 32 bytes.

Another implementation was proposed by Alan Mycroft of Codemist software on the USENET (public domain) network. Moving more than one byte at a time in the innermost loop reduces execution time. If you take a longword with variable name l, consisting of four ASCII characters ($0 to $FF), the C expression

```
(l - 0x01010101L) & ((~l) & 0x80808080L)
```

will be non-zero if and only if the variable l contains a null byte. A 68000 assembly language implementation of this concept is as follows:

```
strcpy: MOVEM.L 4 (SP),A0/A1        ;    MPW Doesn't save these
        MOVE.L    A0,D0             ;    Return Value
        MOVE.L    A1,D1             ;    for word / byte
        OR.L      D0,D1             ;    see if
        ANDI.W    #1,D1             ;           odd
        BNE.S     bytes             ;    Do bytes
        MOVE.L    D2,-(SP)          ;    Save D2
longs:  MOVE.L    (A1)+,D2          ;    Get next long
        MOVE.L    D2,D0             ;    Copy
        MOVE.L    D2,D1             ;           Twice
        SUB.L     #$01010101,D0     ;    Sub 1 from each byte
        NOT.L     D1                ;    1's complement
        AND.L     #$80808080,D1     ;    with 80808080
        AND.L     D1,D0             ;    now and
        BNE.S     wbytes            ;    IF NE, has a nullbyte
        MOVE.L    D2,(A0)+          ;    Store destination
        BRA.S     longs             ;    Do some more longs
                                    ;
wbytes: MOVE.L    D2,D0             ;    Copy again
        SWAP      D0                ;    Make it
        LSR.L     #8,D0             ;           a byte
        LSL.L     #8,D2             ;    Shift to position
        MOVE.B    D0,(A0)+          ;    Store it
        BNE.S     wbytes            ;    And around again
        MOVE.L    (SP)+,D2          ;    Restore
        MOVE.L    4(SP),D0          ;    Return value
        RTS                         ;    Back to caller
                                    ;
bytes:  MOVE.B    (A1)+,(A0)+       ;    Copy a byte
        BNE.S bytes                 ;    Loop
        RTS                         ;    Done
```

Note the immediate data values within the loop (starting at label longs:). The 68020 and 68030 have an instruction cache, that eliminates the need to fetch the instructions from main memory more than once. For a 68010 or 68000, putting these values in registers may be more effective since the instructions must be refetched each time through the loop.

This routine has execution characteristics as follows:

```
Bytes = 0001 times: 015.33 / 017.34 (-13%)
Bytes = 0002 times: 016.17 / 018.66 (-15%)
Bytes = 0004 times: 018.00 / 018.33 (-1%)
Bytes = 0008 times: 021.67 / 021.33 (01%)
Bytes = 0016 times: 028.83 / 026.84 (06%)
Bytes = 0032 times: 043.50 / 038.00 (12%)
Bytes = 0064 times: 072.33 / 060.50 (16%)
Bytes = 0128 times: 130.33 / 105.00 (19%)
Bytes = 0256 times: 247.17 / 194.33 (21%)
Bytes = 0512 times: 480.17 / 372.67 (22%)
```

Note that this routine's performance is poorer than the single-character routine in C for byte sizes less than 8. However, the 22 percent improvement at the higher range is significant when compared with the single-character assembly language routine.

The length distribution of the strings in your application determines which of the two assembly language routines you should use. To put the 4-character routine in an application where most strings are very short is just as foolish as putting the 1-character routine in an application where most strings are long. This need to match software structure to workload is a very common consideration in performance optimization.

strcmp() Another very common routine is the strcmp (string compare) function in the C library. This function compares two strings with the following calling sequence:

```
int    ret,strcmp();          /* Return value is an int          */
char   *a,*b;                 /* -> strings to compare           */

ret = strcmp(a,b);            /* This is the calling sequence    */
```

The return value is negative if a is less than b, zero if a equals b, and positive if a is greater than b. Most implementations return the difference of the first character that does not match, as in the following implementation:

```
strcmp: MOVEM.L   4(SP),A0/A1       ;   Fetch A0/A1
cloop:  MOVE.B    (A0)+,D0          ;   Move,
        BEQ.S     cend              ;   EQ means done
        SUB.B     (A1)+,D0          ;   Subtract
        BEQ.S     cloop             ;   Keep comparing
        EXTB.L    D0                ;020Extend D0 to long
        RTS                         ;   Done
cend:   SUB.B     (A1)+,D0          ;   One last time
        EXTB.L    D0                ;020Extend D0 to long
        RTS                         ;   Done.
```

This implementation was derived in much the same manner as strcpy(). In the interest of brevity, we do not list the C version or the generated code. Note that the EXTB instruction must be recoded as an EXT.W/EXT.L pair on a 68000 or 68010 chip. Again, the registers not preserved by the MPW calling sequence are shamelessly exploited.

The four-character version of this routine is as follows:

```
strcmp: MOVEM.L   4(SP),A0/A1       ;   MPW Doesn't save these
        MOVE.L    A0,D0             ;   Destination Value
        MOVE.L    A1,D1             ;   for word / byte
        OR.L      D0,D1             ;   see if
        ANDI.W    #1,D1             ;          odd
        BNE.S     cbytes            ;   Do bytes
        MOVE.L    D2,-(SP)          ;   Save D2
clongs: MOVE.L    (A1)+,D2          ;   Get next long
        MOVE.L    D2,D0             ;   Copy
        MOVE.L    D2,D1             ;          Twice
        SUB.L     #$01010101,D0     ;   Sub 1 from each byte
        NOT.L     D1                ;   1's complement
        AND.L     #$80808080,D1     ;   with 80808080
        AND.L     D1,D0             ;   now and
        BNE.S     cwbytes           ;   IF NE, has a nullbyte
        CMP.L     (A0)+,D2          ;   Store destination
        BEQ.S     clongs            ;   Do some more longs
                                    ;
cwbytes SUBQ.L    #4,A1             ;   A1 -> Back
        MOVE.L    D2,D0             ;   Copy again
        SWAP      D0                ;   Make it
        LSR.L     #8,D0             ;          a byte
        LSL.L     #8,D2             ;   Shift to position
        SUB.B     (A1)+,D0          ;   Store it
        BNE.S     cwbytes           ;   And around again
        EXTB.L    D0                ;020Extend to long
        MOVE.L    (SP)+,D2          ;   Restore
        RTS                         ;   Back to caller
                                    ;
cbytes  MOVE.B    (A0)+,D0          ;   Move,
        BEQ.S     cend              ;   EQ means done
        SUB.B     (A1)+,D0          ;   Subtract
        BEQ.S       cbytes              ;    Keep comparing
        EXTB.L      D0                  ;020 Extend D0 to long
        RTS                             ;    Done
                                        ;
cend    SUB.B       (A1)+,D0            ;    One last time
        EXTB.L      D0                  ;020 Extend D0 to long
        RTS                             ;    Done.
```

Again, note the use of EXTB.L, which is 68020/68030 dependent. Both the 1-character and 4-character routines show performance similar to that experienced with string copy.

Effect on Dhrystone Benchmark The Dhrystone benchmark was originally designed for the Ada language. It was published by Rheinhold P. Weicker in *Communications of the ACM* in October 1984. The version we use here (version 2.1, dated 05/17/88) was translated into C and posted on the USENET network by Rick Richardson.

Results of running Dhrystone with this small amount of optimization are as follows:

String Routines	Dhrystones/sec	% Improvement
MPW C 3.0 Library	2158.3	0.0%
Single character C	2608.7	20.9%
Single character ASM	2777.8	28.7%
Four character ASM	2803.7	29.9%

The column MPW C 3.0 Library is the result of linking Dhrystone and running it with the character routines supplied. The Single character C column uses the C routines from which the 1-character assembler routines above were derived. Note how much improvement can be made with just a bit of effort. This situation has an obvious implication for C compiler writers: strcpy() and strcmp() should be optimized in any commercial product.

Loop Unrolling Examples

Another area that bears exploration is unrolling loops in critical program areas. We will consider three such routines: one that copies a number of bytes from one area to another, a second routine that clears an area of memory, and a third that compares two memory areas. These functions are used in a large percentage of major software efforts.

Single-byte Versions Code written to perform these functions in the simplest fashion works well for small amounts of data. The single character routines are fairly compact and are discussed below. The C calling sequence for these routines is as follows:

```
char   *src,*dest;                       /*-> Source and destination          */
int    count;                            /*= number of bytes                  */
int    _memcmp(),result;                 /*= comparison result (-1,0,+1)      */

_memcpy(src,dest,count);                 /* Copy count bytes src to dest      */
_memclr(dest,count);                     /* Clear count bytes at dest         */
result = _memcmp(src,dest,count);        /* Compare src to dest for count     */
```

The leading underscore characters (_) eliminate a name conflict with the MPW routines of the same names without the underscores. The comparison result is –1 if the source is less than the destination, 0 if the source equals the destination, and +1 if the source is greater than the destination. This is compatible with the strcmp() routine described above.

The memory copy routine copies a string of bytes terminated by count. A single character implementation of this routine is as follows:

```
_memcpy: MOVEM.L    4(SP),A0/A1        ;    Source / destination
         MOVE.L     12(SP),D0          ;    Count
         BEQ.S      cpydone            ;    A wise guy!
cpyloop: MOVE.B     (A0)+,(A1)+        ;    Move a long
         SUBQ.L     #1,D0              ;    Count
         BNE.S      cpyloop            ;         down
cpydone:RTS                            ;    Return to caller
```

Note the use of A0,A1, and D0, which need not be preserved across a function call. Note also that the routine does not use the DBRA instruction, which would limit the operation to 65,536 bytes. Handling the case where the count was specified as zero requires a special instruction. With a count of zero, the routine would attempt to move the entire 32-bit memory space (4Gb), which is obviously ridiculous. You could make a case for changing the BEQ/BNE instructions to BLE instructions, which would disallow a negative count. This applies to the next two code fragments as well.

The memory clear routine is similar:

```
_memclr: MOVE.L     4(SP),A0           ;    -> Area to clear
         MOVE.L     8(SP),D0           ;    = count
         BEQ.S      clrdone            ;    A wise guy!
clrloop: CLR.B      (A0)+              ;    Clear a longword
         SUBQ.L     #1,D0              ;    Count down
         BNE.S      clrloop            ;    And Loop
clrdone: RTS                           ;    Done
```

The memory comparison routine must return a value in register D0. This implementation is

```
_memcmp: MOVEM.L   4(SP),A0/A1       ;    Source / destination
         MOVE.L    12(SP),D0         ;    Count
         BEQ.S     cmpdone           ;    A wise guy!
cmploop: CMP.B     (A0)+,(A1)+       ;    Compare a byte
         BNE.S     cmpret            ;    Set return value
         SUBQ.L    #1,D0             ;    Decrement
         BNE.S     cmploop           ;    And loop
cmpdone: RTS                         ;    Done.

cmpret:  BMI.S     cmpless           ;    a < b
         MOVEQ.L   #1,D0             ;    > 0; a > b
         RTS                         ;    Return to caller
cmpless: MOVEQ.L   #-1,D0            ;    a < b
         RTS                         ;    Return
```

If the two memory areas are equal, D0 counts down to zero, which is the correct return value. Note that this routine considers zero-length areas to be equal.

The timings on these routines are as follows:

Size	**clear**	**copy**	**compare**
00001	0007.26	0008.93	0010.56
00002	0004.18	0005.01	0005.84
00004	0002.36	0003.06	0003.75
00008	0001.59	0002.01	0002.71
00016	0001.18	0001.52	0002.15
00032	0000.97	0001.30	0001.87
00064	0000.87	0001.16	0001.75
00128	0000.82	0001.09	0001.68
00256	0000.80	0001.06	0001.65
00512	0000.78	0001.05	0001.64
01024	0000.78	0001.04	0001.23
02048	0000.78	0001.04	0001.23
04096	0000.78	0001.04	0001.24

Sizes are in bytes and the times are in microseconds per byte. As in the string illustration, the time per byte decreases to a threshold value as the size increases. The innermost loop dominates execution time at about 128 bytes for the clear and copy routines and 1024 bytes for the comparison routine.

Unrolled Versions As with the string routines, one way to optimize these functions is to use long word (4 byte) operations wherever possible, resorting to byte operations only if the data or count is misaligned. Another technique is to reduce the overhead involved in the loop instructions. This section presents fully unrolled versions of the copy, clear, and comparison routines.

The unrolled version of the copy routine is as follows:

```
SIZEL   EQU   32
SIZEB   EQU   8
_memcpy MOVEM.L   4(SP),A0/A1          ;      Source / destination
        MOVE.L    12(SP),D0            ;      Count
        MOVE.L    D0,D1                ;      A scratch register
        CMP.L     #SIZEB,D1            ;      See if too small
        BLE.S     cpybyte              ;      Just copy bytes.
        OR.L      4(SP),D1             ;      Or in
        OR.L      8(SP),D1             ;            all args
        ANDI.B    #3,D1                ;      Long aligned?
        BNE.S     cpybyte              ;      No.
        MOVEQ.L   #SIZEL,D1            ;      Put size in register
cpyloop SUB.L     D1,D0                ;      Subtract size 1st
        BMI.S     cpyrest              ;      Copy residual
        MOVE.L    (A0)+,(A1)+          ;      Copy a long
        MOVE.L    (A0)+,(A1)+          ;      Copy a long
        MOVE.L    (A0)+,(A1)+          ;      Copy a long
        MOVE.L    (A0)+,(A1)+          ;      Copy a long
        MOVE.L    (A0)+,(A1)+          ;      Copy a long
        MOVE.L    (A0)+,(A1)+          ;      Copy a long
        MOVE.L    (A0)+,(A1)+          ;      Copy a long
        MOVE.L    (A0)+,(A1)+          ;      Copy a long
        BRA.S     cpyloop              ;      And loop
cpyrest ADD.L     D1,D0                ;      Re-add size
        BNE.S     cpylong              ;      EQ means done.
        RTS                            ;      back
cpylong MOVE.L    (A0)+,(A1)+          ;      Move a long
        SUBQ.L    #1,D0                ;      Count
        BNE.S     cpylong              ;            down
cpycomp RTS                            ;      Return to caller
```

The SIZEL and SIZEB equates give the number of bytes copied by the innermost loops. By varying these equates and the number of instructions in the loops, you can tune the performance of the routines to reflect the size distribution in a particular application. These particular values were chosen to optimize the performance discussed below.

The instructions at the beginning of the procedure determine whether the parameters dictate a longword or byte copy operation. To qualify for a longword copy, both source and destination must be aligned on longword boundaries, and the count must be a multiple of four. If the count is less than the size of a byte copy, a byte copy is performed regardless of alignment. This helps performance on short copy operations.

The main copy loop begins at label cpyloop. Loop termination is done by the first two instructions in the loop. If fewer than the number of bytes specified by the SIZEL equate remain, the copy proceeds one longword at a time (at label cpyrest). This allows copy lengths that are not an even multiple of the size moved by the inner loop. There are three overhead instructions in this loop (the SUBQ/BMI pair at the beginning, and the BRA at the end). Note that 32 bytes are moved each time these instructions execute, thus reducing the overhead to an acceptable level.

The loop terminates when subtracting the length of the main loop from the count yields a negative result. Code that completes the copy must first add this length back to the count register (D0) and copy one longword at a time until the count hits zero.

The label cpybyte begins the section of the _memcpy routine, which copies with byte instructions rather than longword instructions. This code is as follows:

```
cpybyte SUBQ.L    #SIZEB,D0        ;      Subtract size 1st
        BMI.S     cpyres           ;      Copy residual
        MOVE.B    (A0)+,(A1)+      ;      Move a byte
        MOVE.B    (A0)+,(A1)+      ;      Move a byte
        MOVE.B    (A0)+,(A1)+      ;      Move a byte
        MOVE.B    (A0)+,(A1)+      ;      Move a byte
        MOVE.B    (A0)+,(A1)+      ;      Move a byte
        MOVE.B    (A0)+,(A1)+      ;      Move a byte
        MOVE.B    (A0)+,(A1)+      ;      Move a byte
        MOVE.B    (A0)+,(A1)+      ;      Move a byte
        BRA.S     cpybyte          ;      And around again
cpyres  ADDQ.L    #SIZEB,D0        ;      Restore
        BEQ.S     cpycomp          ;      Completed.
cpy1    MOVE.B    (A0)+,(A1)+      ;      1 / time
        SUBQ.L    #1,D0            ;      And
        BNE.S     cpy1             ;             loop
        RTS                        ;      Finished.
```

Note that this code fragment uses the same techniques as before. The one exception is the loop length and the fact that the SUBQ/ADDQ instructions take the place of keeping the count in a data register. If the innermost loop is larger than 8 bytes, you must use the technique above, because the SUBQ and ADDQ instructions have an immediate value in the range 1 to 8.

The memory clear routine follows:

```
_memclr:MOVE.L    4(SP),A0             ;      -> Area to clear
        MOVE.L    8(SP),D0             ;      = count
        MOVE.L    D0,D1                ;      Copy count
        CMP.L     #SIZEB,D1            ;      See if too small
        BLE.S     clrbyte              ;      Just copy bytes.
        OR.L      4(SP),D1             ;      and OR
        ANDI.B    #3,D1                ;      Long aligned?
        BNE.S     clrbyte              ;      Clear bytes
        MOVEQ.L   #SIZEL,D1            ;      Put size in register
clrloop:SUB.L     D1,D0                ;      Subtract size 1st
        BMI.S     clrrest              ;      Copy residual
        CLR.L     (A0)+                ;      Clear a long
        CLR.L     (A0)+                ;      Clear a long
        CLR.L     (A0)+                ;      Clear a long
        CLR.L     (A0)+                ;      Clear a long
        CLR.L     (A0)+                ;      Clear a long
        CLR.L     (A0)+                ;      Clear a long
        CLR.L     (A0)+                ;      Clear a long
        CLR.L     (A0)+                ;      Clear a long
        BRA.S     clrloop              ;      And loop
clrrest:ADD.L     D1,D0                ;      Re-add size
        BNE.S     clrlong              ;      EQ means done.
        RTS                            ;      back
clrlong:CLR.L     (A0)+                ;      Move a long
        SUBQ.L    #1,D0                ;      Count
        BNE.S     clrlong              ;            down
clrcomp:RTS                            ;      Return to caller
                                       ;
clrbyte:SUBQ.L    #SIZEB,D0            ;      Subtract size 1st
        BMI.S     clrres               ;      Copy residual
        CLR.B     (A0)+                ;      Clear a byte
        CLR.B     (A0)+                ;      Clear a byte
        CLR.B     (A0)+                ;      Clear a byte
        CLR.B     (A0)+                ;      Clear a byte
        CLR.B     (A0)+                ;      Clear a byte
        CLR.B     (A0)+                ;      Clear a byte
        CLR.B     (A0)+                ;      Clear a byte
        CLR.B     (A0)+                ;      Clear a byte
        BRA.S     clrbyte              ;      And around again
clrres: ADDQ.L    #SIZEB,D0            ;      Restore
        BEQ.S     clrcomp              ;      Completed.
clr1:   CLR.B     (A0)+                ;      1 / time
        SUBQ.L    #1,D0                ;      And
        BNE.S     clr1                 ;            loop
        RTS                            ;      Finished.
```

This procedure is similar to the copy procedure except that there is only one address parameter and the instruction to be unrolled is a

CLR and not a MOVE. Comparison, on the other hand, is a bit different. This function is shown below:

```
_memcmp:MOVEM.L   4(SP),A0/A1            ;     Source / destination
        MOVE.L    12(SP),D0              ;     Count
        MOVE.L    D0,D1                  ;     A scratch register
        CMP.L     #SIZEB,D1              ;     See if too small
        BLE.S     cmpbyte                ;     Just compare bytes.
        OR.L      4(SP),D1               ;     Or in
        OR.L      8(SP),D1               ;            all args
        ANDI.B    #3,D1                  ;     Long aligned?
        BNE.S     cmpbyte                ;     No.
        MOVEQ.L   #SIZEL,D1              ;     Put size in register
cmploop:SUB.L     D1,D0                  ;     Subtract size 1st
        BMI.S     clrrest                ;     Compare residual
        CMP.L     (A0)+,(A1)+            ;     Compare a long
        BNE.S     cmpret                 ;     Not equal, quit
        CMP.L     (A0)+,(A1)+            ;     Compare a long
        BNE.S     cmpret                 ;     Not equal, quit
        CMP.L     (A0)+,(A1)+            ;     Compare a long
        BNE.S     cmpret                 ;     Not equal, quit
        CMP.L     (A0)+,(A1)+            ;     Compare a long
        BNE.S     cmpret                 ;     Not equal, quit
        CMP.L     (A0)+,(A1)+            ;     Compare a long
        BNE.S     cmpret                 ;     Not equal, quit
        CMP.L     (A0)+,(A1)+            ;     Compare a long
        BNE.S     cmpret                 ;     Not equal, quit
        CMP.L     (A0)+,(A1)+            ;     Compare a long
        BNE.S     cmpret                 ;     Not equal, quit
        CMP.L     (A0)+,(A1)+            ;     Compare a long
        BNE.S     cmpret                 ;     Not equal, quit
        BRA.S     cmploop                ;     And loop
cmprest:ADD.L     D1,D0                  ;     Re-add size
        BNE.S     cmplong                ;     EQ means done.
        RTS                              ;     back
cmplong:CMP.L     (A0)+,(A1)+            ;     Compare a long
        BNE.S     cmpret                 ;     Finished.
        SUBQ.L    #1,D0                  ;     Count
        BNE.S     cmplong                ;            down
cmpcomp:RTS                              ;     Return to caller
                                         ;
cmpret: BMI.S     cmpless                ;     a < b
        MOVE.L    #1,D0                  ;     > 0; a > b
        RTS                              ;     Return to caller
cmpless:MOVE.L    #-1,D0                 ;     a < b
        RTS                              ;     Return
```

The important difference in this function is that the CMP and BNE instructions must be unrolled together, since a difference at any point means that the loop must terminate. The byte comparison part of this routine follows:

```
cmpbyte:SUBQ.L     #SIZEB,D1          ;     Subtract size 1st
        BMI.S      clrres             ;     Compare residual
        CMP.B      (A0)+,(A1)+        ;     Compare a byte
        BNE.S      cmpret             ;     Not equal, quit
        CMP.B      (A0)+,(A1)+        ;     Compare a byte
        BNE.S      cmpret             ;     Not equal, quit
        CMP.B      (A0)+,(A1)+        ;     Compare a byte
        BNE.S      cmpret             ;     Not equal, quit
        CMP.B      (A0)+,(A1)+        ;     Compare a byte
        BNE.S      cmpret             ;     Not equal, quit
        CMP.B      (A0)+,(A1)+        ;     Compare a byte
        BNE.S      cmpret             ;     Not equal, quit
        CMP.B      (A0)+,(A1)+        ;     Compare a byte
        BNE.S      cmpret             ;     Not equal, quit
        CMP.B      (A0)+,(A1)+        ;     Compare a byte
        BNE.S      cmpret             ;     Not equal, quit
        CMP.B      (A0)+,(A1)+        ;     Compare a byte
        BNE.S      cmpret             ;     Not equal, quit
        BRA.S      cmpbyte            ;     And around again
cmpres: ADDQ.L     #SIZEB,D0          ;     Restore
        BEQ.S      cmpcomp            ;     Completed.
cmp1:   CMP.B      (A0)+,(A1)+        ;     1 / time
        BNE.S      cmpret             ;     Not equal, quit
        SUBQ.L     #1,D0              ;     And
        BNE.S      cmp1               ;           loop
        RTS                           ;     Finished.
```

Note that the 68000 big endian byte ordering (highest byte at lowest address) means that character, word, and longword comparisons will all work with this code. On a machine with a different byte ordering, you need a parameter to specify the size of the items to be compared.

You would expect the comparison operation to improve roughly half as much as the other two, due to the doubling of instructions inside the loop. The presence of the overhead code at the front of the routine implies that the performance will be much better with larger memory area sizes. These routines perform as follows:

Size	**clear**	**copy**	**compare**
00001	0008.89	0011.16	0016.69
00002	0005.01	0005.84	0008.63
00004	0002.78	0003.48	0004.72
00008	0001.67	0002.01	0004.09
00016	0001.38	0001.87	0001.53
00032	0000.45	0000.67	0000.69
00064	0000.29	0000.44	0000.49
00128	0000.22	0000.31	0000.39
00256	0000.18	0000.26	0000.34
00512	0000.16	0000.22	0000.32
01024	0000.15	0000.21	0000.31
02048	0000.15	0000.20	0000.30
04096	0000.14	0000.20	0000.30

A quick glance shows that the unrolled routines are slower than the single character routines with byte counts below 32 bytes. After 2048 bytes, the unrolled routines are 5.6 times faster for clear, 5.1 times faster for copy, and 4.1 times faster for compare.

qsort

Another compute-intensive process is sorting items in main memory. We present here an optimized version of the quicksort algorithm described by Kernighan and Plauger in *Software Tools in Pascal* (Addison-Wesley, 1981). Most C language systems support a version of this algorithm using a function called qsort. This routine has the following calling conventions:

```
void   qsort();
void   *base;                  /* -> Base of array to sort          */
int    nmemb;                  /* = Number of records to sort       */
int    size;                   /* = Number of bytes per record      */
int    (*compar)();            /* Compares two records              */

qsort(base,nmemb,size,compar); /* Sort array at base                */
```

The function sorts a contiguous array of records at the address specified by the base parameter. Each record contains size bytes, and there are nmemb of them. The array is thus bounded by addresses base and base+(size*nmemb).

Sorting is controlled by a function passed as a parameter. This function has the following calling sequence:

```
void   *a,*b;                  /* -> to two records to compare     */
int    compar();               /* function to compare them         */
int    result;                 /* Result of comparison             */

result = compar(a,b);          /* Do the comparison                */
```

The result of the comparison is compatible with the strcmp() function described earlier in this chapter. The result is less than zero if a is less than b, zero if a is equal to b, and greater than zero if a is greater than b.

qsort() works by physically moving the records in main memory. For large records, you may wish to consider constructing an array of pointers and sorting this pointer array. The comparison function mechanism gives you the freedom to structure the sort in this manner.

C Implementation The quicksort algorithm was developed by C. A. R. Hoare and is recursive in nature. The implementation presented here attempts to minimize the procedure call overhead by using an inner procedure with only those parameters which are absolutely required. The C version of this routine is as follows:

```
void qsort(base,nmemb,size,compar)       /***********************************/
void *base;                              /* Base of array to sort           */
long nmemb;                              /* Number of array elements        */
long size;                               /* Size of each array elements     */
int  compar(void *a,void *b);            /* Comparison function             */
{                                        /*                                 */
  _qsort((char *)base,                   /* Call real function              */
          (char *)(long)base + (size*(nmemb-1)), /* End item                */
          (unsigned long)size,compar);   /* size and comparison function    */
}                                        /***********************************/
```

This first part of the function simply converts the input parameters to a form more suitable for use in a recursive routine. The algorithm requires the address of the first and last members of the array. In order to eliminate the multiply operation on each call, a separate recursive routine is required. This routine has the following C implementation:

```
void _qsort(lo,hi,size,compar)           /* real work done here             */
char *lo;                                /* -> First element                */
char *hi;                                /* -> Last element                 */
unsigned long size;                      /* = # of bytes per element        */
int (*compar)(void *a,void *b);          /* Comparison function             */
{                                        /*                                 */
  char *i,*j;                            /* Temporaries                     */
  char *pivot;                           /* Pivot position                  */
                                         /*-------------------------------- */
       if(hi <= lo) return;              /* Just 1, it's sorted             */
       i = lo;                           /* i -> First element              */
       j = hi;                           /* j -> Last element               */
       pivot = hi;                       /* Pivot initially at end          */
                                         /*-------------------------------- */
       while(i < j)                      /* Until looked at all elements    */
       {                                 /*                                 */
         while((i < j)             &&    /* skip initial ordered pairs      */
              ((*compar)(i,pivot) <= 0)) /*                                 */
               i += size;                /* Bump to next element            */
         while((j >i)              &&    /* skip final ordered pairs        */
              ((*compar)(j,pivot) >= 0)) /*                                 */
               j -= size;                /* Bump to prev element            */
         if(i < j)                       /* Out of order pair               */
           exchange(i,j,size);           /* Exchange them                   */
       }                                 /* until done                      */
```

```
                                    /*--------------------------------*/
exchange(i,hi,size);                /* Move pivot to i                */
if((i - lo) < (hi - i))             /* Pick partition size            */
{                                   /*                                */
  _qsort(lo,i-size,size,compar);    /* From low end                   */
  _qsort(i+size,hi,size,compar);    /* Then from high                 */
}                                   /*                                */
else                                /* Do it the other way            */
{                                   /*                                */
  _qsort(i+size,hi,size,compar);    /* from high end                  */
  _qsort(lo,i-size,size,compar);    /* Then from low                  */
}                                   /*                                */
                                    /*********************************/
```

The way this algorithm works is, to use a writer's cliché, "adequately described in the literature." Our purpose here is to show how performance can be enhanced using assembly language. You will note that the _qsort() routine calls two external functions, exchange(), and compar(). This latter function is supplied by the caller of qsort(), as described above. The exchange() function simply exchanges two data areas for a given number of bytes. This can be trivially derived from the memory copy routine described earlier.

Note the declaration of the i, j, and pivot variables. Since these quantities are never used as pointers, they are declared char * so that they can be used essentially as integers. This declaration also keeps the compiler from complaining about mismatched types when calling the compar() function.

Assembly Language Implementation There are three important considerations in the C implementation. First, the size and compar parameters are constant. In assembly language, you could dedicate a register for each of these parameters. Second, the exchange() function is called only from within _qsort(). You might be able to optimize this operation with a non-standard calling sequence. Third, the compar() function requires standard C calling conventions and must be allowed to destroy registers not preserved by the calling sequence.

The function qsort() in this implementation must convert from the C calling convention to the specialized calling conventions for _qsort(). This function has the following implementation:

```
qsort                                   ;       Here to sort
        LINK      A6,#0                 ;       Preserve framing
        MOVEM.L   D2-D7/A4,-(SP)        ;       Save registers
        MOVEM.L   8(A6),D2-D3/D7/A4     ;       4 args
        SUBQ.L    #1,D3                 ;       Subtract 1
        MULS.L    D7,D3                 ;020    D3 = size*(nmemb-1)
        ADD.L     D2,D3                 ;       Add in base to get hi
        BSR.S     _qsort                ;       Call recursive routine
        MOVEM.L   (SP)+,D2-D7/A4        ;       Pop registers
        UNLK      A6                    ;       Unlink stack frame
        RTS                             ;       And begone!
```

Note the 32-bit multiply, which is not implemented on the 68000 or 68010. If you are sorting areas where both the size and the number of records will fit into 16 bits, you can use the 68000 MULS.W instruction. The non-standard calling sequence used for the _qsort() function places all values in registers. We use D2–D7 and A4 (which are preserved by the C calling standard). These registers are assigned as follows:

Register	Usage
D2	lo parameter to _qsort
D3	hi parameter to _qsort
D4	i internal variable inside _qsort
D5	j internal variable inside _qsort
D6	pivot internal variable inside _qsort
D7	size parameter to _qsort
A4	compar parameter to _qsort

It is important to note that the choices for D4–D6 allow a single MOVEM instruction to set up the parameters to the user-supplied compar() function.

_qsort has the following implementation:

```
_qsort  CMP.L     D2,D3                 ;       Hi : Lo
        BLE       qscomp                ;       LE means done
        MOVE.L    D2,D4                 ;       i = lo
        MOVE.L    D3,D5                 ;       j = hi
        MOVE.L    D3,D6                 ;       pivot = hi
                                        ;
qs10    CMP.L     D4,D5                 ;       while (i < j)
        BLE.S     qs50                  ;       CMP is inverted, remember...
                                        ;
qs20    BRA.S     qs22                  ;       Skip initial
qs21    ADD.L     D7,D4                 ;       i += size;
```

```
qs22    CMP.L     D4,D5             ;      while ((i < j) &&
        BLE.S     qs32              ;
        MOVEM.L   D4/D6,-(SP)       ;      Push args
        JSR       (A4)              ;      call function
        ADDQ.L    #8,SP             ;      Pop args
        TST.L     D0                ;      compar(i,pivot) <= 0))
        BLE.S     qs21              ;      if not LE, done with while
                                    ;
qs30    BRA.S     qs32              ;      Skip loop body
qs31    SUB.L     D7,D5             ;      j -= size;
qs32    CMP.L     D4,D5             ;      while ((j > i) &&
        BLE.S     qs40              ;
        MOVEM.L   D5/D6,-(SP)       ;      Push args
        JSR       (A4)              ;      call function
        ADDQ.L    #8,SP             ;      Pop args
        TST.L     D0                ;      compar(j,pivot) >= 0))
        BGE.S     qs31              ;      Done with while
                                    ;
qs40    CMP.L     D4,D5             ;      if(i < j) out of order
        BLE.S     qs10              ;      In order, and OK
        MOVE.L    D4,A0             ;      Set
        MOVE.L    D5,A1             ;        up
        MOVE.L    D7,D0             ;             parameters
        BSR.S     exchange          ;      exchange(i,j,size);
        BRA.S     qs10              ;      End of while(i < j)
                                    ;
qs50    MOVE.L    D4,A0             ;      exchange(i,hi,size);
        MOVE.L    D3,A1             ;
        MOVE.L    D7,D0             ;
        BSR.S     exchange          ;
                                    ;
qs60    MOVE.L    D4,D0             ;      i  -
        SUB.L     D2,D0             ;             lo
        MOVE.L    D3,D1             ;      hi -
        SUB.L     D4,D1             ;             i
        CMP.L     D0,D1             ;      if(i-lo < hi - i)
        BLE.S     qs70              ;      it isn't.
                                    ;
        MOVEM.L   D2/D3/D4,-(SP)    ;      Need to save lo,hi,i
        MOVE.L    D4,D3             ;      replace hi with
        SUB.L     D7,D3             ;      i - size
        BSR.S     _qsort            ;      qsort(lo,i-size,...);
        MOVEM.L   (SP),D2/D3/D4     ;      Restore the registers
        MOVE.L    D4,D2             ;      Replace lo with
        ADD.L     D7,D2             ;      i + size
        BSR.S     _qsort            ;      _qsort(i+size,hi,..);
        MOVEM.L   (SP)+,D2/D3/D4    ;      Restore original registers
        RTS                             ;       And return
                                        ;
qs70    MOVEM.L   D2/D3/D4,-(SP)        ;       Need to save lo,hi,i
        MOVE.L    D4,D2                 ;       Replace lo with
        ADD.L     D7,D2                 ;       i + size
        BSR.S     _qsort                ;       _qsort(i+size,hi,..);
        MOVEM.L   (SP),D2/D3/D4         ;       Restore the registers
        MOVE.L    D4,D3                 ;       replace hi with
        SUB.L     D7,D3                 ;       i - size
        BSR       _qsort                ;       qsort(lo,i-size,...);
        MOVEM.L   (SP)+,D2/D3/D4        ;       Restore original registers
qscomp  RTS                             ;       And return
```

This is a relatively straightforward translation from C to assembler, given the register conventions described above. There are three while loops in the C code, two of which are nested inside a larger loop. These innermost loops are located between labels qs20 to qs30, and qs30 to qs40. These loops start with a branch instruction. A standard compiler trick is to place the loop body before testing the termination condition and to branch to the tests initially. This causes the body of the loop to contain one less branch instruction.

Note that the parameters to the compar() function are set up with a single MOVEM instruction. The register assignment was critical here for two reasons. First, the comparison routine is free to trash registers not preserved by the calling sequence standard. Since we did not use these registers globally there is no need to save and restore them. Second, the choice of D4–D6 caused the parameters to be pushed on the stack in the proper order by the MOVEM instruction.

Calls to the exchange() function (non-standard calling sequence) occur between labels qs40 and qs60. This function has the following implementation:

```
SIZEL    EQU       32
SIZEB    EQU       8
exchange                                  ;      Here to exchange
         MOVE.L    D2,-(SP)               ;      Save a temporary
         CMP.L     #SIZEB,D0              ;      See if too small
         BLE.S     excbyte                ;      Just exchange bytes.
         MOVE.L    A0,D2                  ;      See
         OR.L      D0,D2                  ;         if
         MOVE.L    A1,D1                  ;             Longs
         OR.L      D1,D2                  ;                   OK
         ANDI.B    #3,D2                  ;      Long aligned?
         BNE.S     excbyte                ;      No.
         MOVEQ.L   #SIZEL,D1              ;      Put size in register
excloop  SUB.L     D1,D0                  ;      Subtract size 1st
         BMI.S     excrest                ;      Exchange residual
                                          ;0
         MOVE.L    (A0),D2                ;      Fetch
         MOVE.L    (A1),(A0)+             ;             and
         MOVE.L    D2,(A1)+               ;                   exchange
                                          ;1
         MOVE.L    (A0),D2                ;      Fetch,
         MOVE.L    (A1),(A0)+             ;             and
         MOVE.L    D2,(A1)+               ;                   exchange
                 .
                 .
                 .
                                          ;7
         MOVE.L    (A0),D2                ;      Fetch,
         MOVE.L    (A1),(A0)+             ;             and
         MOVE.L    D2,(A1)+               ;                   exchange
                                          ;
         BRA.S     excloop                ;      And loop
excrest  ADD.L     D1,D0                  ;      Re-add size
         BNE.S     exclong                ;      EQ means done.
         MOVE.L    (SP)+,D2               ;      Restore D2
         RTS                              ;      back
exclong  MOVE.L    (A0),D2                ;      Fetch,
         MOVE.L    (A1),(A0)+             ;             and
         MOVE.L    D2,(A1)+               ;                   exchange
         SUBQ.L    #4,D0                  ;      Count
         BNE.S     exclong                ;             down
exccomp  MOVE.L    (SP)+,D2               ;      Fix what we broke
         RTS                              ;      Return to caller
```

Note the similarity to the block copy function described earlier. The source and destination addresses are passed in registers A0 and A1 and the byte count is passed in register D0. There are eight sets of move instructions in the source, some of which are not shown above. Note that register D2 is saved and restored and used as a temporary location for the copied data.

The byte exchange code is also similar to preceding examples.

```
excbyte SUBQ.L    #SIZEB,D0       ;      Subtract size 1st
        BMI.S     excres          ;      Copy residual
                                  ;0
        MOVE.B    (A0),D2         ;      Fetch
        MOVE.B    (A1),(A0)+      ;             and
        MOVE.B    D2,(A1)+        ;                   exchange
                                  ;1
        MOVE.B    (A0),D2         ;      Fetch
        MOVE.B    (A1),(A0)+      ;            and
        MOVE.B    D2,(A1)+        ;                   exchange
               .
               .
               .
                                  ;7
        MOVE.B    (A0),D2         ;      Fetch
        MOVE.B    (A1),(A0)+      ;            and
        MOVE.B    D2,(A1)+        ;                   exchange
                                  ;
        BRA.S     excbyte         ;      And around again
excres  ADDQ.L    #SIZEB,D0       ;      Restore
        BEQ.S     exccomp         ;      Completed.
exc1    MOVE.B    (A0),D2         ;      Fetch,
        MOVE.B    (A1),(A0)+      ;            and
        MOVE.B    D2,(A1)+        ;                   exchange
        SUBQ.L    #1,D0           ;      And
        BNE.S     exc1            ;            loop
        MOVE.L    (SP)+,D2        ;      Fix what we broke
        RTS                       ;      Finished.
```

Performance So what does this effort buy you? The quicksort algorithm exhibits worst-case behavior when the data to be sorted is in inverse order. The performance is $O(n^2)$, or roughly proportional to the number of records squared. Thus, if you double the number of items to be sorted, you would expect the time required to increase by a factor of 4.

Interestingly enough, the record size did not make much difference in the amount of improvement. The following worst-case times were obtained for the Macintosh II:

	12-byte Records			**260-Byte Records**		
Records	**C**	**Asm**	**Gain**	**C**	**Asm**	**Gain**
00032	00009.50	00005.51	42.0%	00007.50	00003.66	51.2%
00064	00032.82	00020.84	36.5%	00029.67	00016.84	43.2%
00128	00123.66	00080.34	35.0%	00117.34	00072.66	38.1%
00256	00478.67	00316.16	34.0%	00465.50	00301.01	35.3%
00512	01882.67	01253.16	33.4%	01857.67	01223.51	34.1%

Times are in milliseconds to sort the given number of records. Larger records offer a slightly higher improvement, probably due to the register-based calling sequence for the exchange() function. Note that this function is in assembly language for both the C and assembly versions of qsort().

bsearch

Searching is an area somewhat related to sorting. Many algorithms for finding a random piece of data inside a table depend on an ordered table. Many C language implementations feature a function called bsearch() that performs a binary search on a sorted table. Like qsort(), the data for a binary search must be a contiguous array of records, each with the same length. The calling sequence for bsearch() is

```
void  *bsearch();                             /* returns a pointer to item      */
void  *rval;                                  /* Return value                   */
void  *key;                                   /* -> element for search          */
void  *base;                                  /* -> beginning of array          */
int    nmemb;                                 /* = Number of records to search  */
int    size;                                  /* = Number of bytes per record   */
int   (*compar)();                            /* Compares two records           */

rval = bsearch(key,base,nmemb,size,compar);
```

The base, nmemb, size, and compar parameters are the same as the qsort() parameters of the same name, described above. The key parameter points to the key of the record to be found. The return value is a pointer to the desired record or zero (NULL) if the record is not in the table.

The algorithm used here is described in Knuth's book, *The Art of Computer Programming, Volume 3: Sorting and Searching* (Addison-Wesley, 1973).

C Implementation An implementation of this algorithm in C is as follows:

```
void *bsearch(key,base,nmemb,size,compar)  /*********************************/
void *key;                                 /* -> record to find             */
void *base;                                /* Base of array to search       */
long nmemb;                                /* Number of array elements      */
long size;                                 /* Size of each array elements   */
int  (*compar)(void *a,void *b);           /* Comparison function           */
{                                          /*                               */
    int  i;                                /* Temp for return value         */
    char *hi,*lo;                          /* Extreme end pointers          */
    char*mid;                              /* Middle pointer                */
    lo = (char *)base;                     /*-> first record                */
    hi = (char *)base + size*(nmemb-1);    /*-> Last record                 */
    while(lo <= hi)                        /* Until pointers cross.         */
    {                                      /*                               */
      mid = (char *)                       /* calculate midpoint as         */
        (((long)hi - (long)lo) /           /* Difference between hi & lo    */
        (size << 1)) * size) + lo;         /* Truncate to record boundary   */
                                           /*                               */
      if(!(i = (*compar)(mid,key)))        /* if compare is zero, then      */
        return (void *)mid;                /*  found it. Return address.    */
      if(i > 0) hi = mid - size;           /* adjust hi boundary            */
      else   lo = mid + size;              /* adjust lo boundary            */
    }                                      /*                               */
    return (void *)0;                      /* Not found                     */
}                                          /*********************************/
```

The routine works like this: Calculate the addresses of the first and last records in the array. Store these in the variables lo and hi. Calculate the midpoint between these two, in variable mid. (This assignment statement is perhaps the most complex in this routine.) Take the difference between hi and lo which is the number of bytes between these two records. Divide this by size times two to get the number of records from the lo record to the middle. Then multiply the quotient of this division by size to get the address of this midpoint record. Since the remainder of an integer division is discarded, this complicated statement guarantees that the mid variable points to the beginning of a record.

You then compare the midpoint record to the record for which you are searching. If they are equal, you found it. If not, move one of the lo or hi variables, and repeat the process. If the hi variable ever becomes less than the lo variable, the desired record is not in the table.

This implementation is designed to work with fixed-length records. If you wish to search a table of variable length records, you must construct an array of pointers to these records and pass this address to the bsearch() routine.

Assembly Language Implementation The C implementation has two variables that are not often referenced: the base and nmemb parameters. The other variables are all referenced within the search loop, making them prime candidates for placement in registers. Since no pointer variables other than the address of the compar routine, are ever de-referenced, you can assign them to data registers if you so desire. (However, since you must multiply and divide the pointer variables, they are assigned to data registers for efficiency.) The resulting implementation looks like this:

```
bsearch: LINK      A6,#0                 ;      Preserve stack frames
         MOVEM.L   D2-D6/A4,-(SP)        ;      Save registers we want
         MOVEM.L   8(A6),D3/D4/D5/D6/A4  ;      key,base,nmemb,size,compar
         SUBQ.L    #1,D5                 ;      Calculate
         MULU.L    D6,D5                 ;      hi = (nmemb-1)*size
         ADD.L     D4,D5                 ;                        + lo
                                         ;      Start while loop
         BRA.S     bs40                  ;      With conditional last
bs10:    MOVE.L    D5,D2                 ;      Mid = hi -
         SUB.L     D4,D2                 ;                lo
         ASR.L     #1,D2                 ;      / 2
         DIVS.L    D6,D2                 ;      / size
         MULS.L    D6,D2                 ;      * size
         ADD.L     D4,D2                 ;      + lo;
         MOVEM.L   D2/D3,-(SP)           ;      Push parms
         JSR       (A4)                  ;      Call compare func.
         ADDQ.L    #8,SP                 ;      Pop stack
         TST.L     D0                    ;      Check sign
         BGT.S     bs20                  ;      mid > key
         BMI.S     bs30                  ;      mid < key
         MOVE.L    D2,D0                 ;      mid = key
         MOVE.L    D0,A0                 ;      For picky C compilers
         MOVEM.L   (SP)+,D2-D6/A4        ;      Pop registers
         UNLK      A6                    ;      Restore A6
         RTS                             ;      And return
                                         ;
bs20:    MOVE.L    D2,D5                 ;      hi = mid -
         SUB.L     D6,D5                 ;                size
         BRA.S     bs40                  ;      Do loop termination code
bs30:    MOVE.L    D2,D4                 ;      lo = mid +
         ADD.L     D6,D4                 ;                size
                                         ;
bs40:    CMP.L     D4,D5                 ;      lo : hi
         BHS.S     bs10                  ;      hi >= lo if true
         MOVEQ.L   #0,D0                 ;      Null pointer
         MOVE.L    D0,A0                 ;      For picky C compilers
         MOVEM.L   (SP)+,D2-D6/A4        ;      Pop registers
         UNLK      A6                    ;      Restore A6
         RTS                             ;      And return
```

The register assignment is probably the trickiest part of the routine. We avoided the registers that might be destroyed by the comparison routine (D0–D1 and A0–A1), making the assignments as follows:

Register	**Usage**
D2	mid variable inside bsearch
D3	key parameter to bsearch
D4	lo variable inside bsearch
D5	hi variable inside bsearch
D6	size parameter to bsearch
A4	compar parameter to bsearch

The major consideration was to get the registers assigned so that parameters could be both fetched and stored with single MOVEM instructions. In particular, the key parameter had to reside in a higher-numbered register than the mid variable so that the call to the comparison function could be done with three instructions (MOVEM, JSR, ADDQ).

The lines before label bs10 perform the initialization for this routine. The base parameter and the nmemb parameter are converted into the lo and hi variables, respectively.

The while loop is coded with the body first, followed by the conditional test. This code is entered by branching to the conditional test, eliminating an unconditional branch at the end of the loop. The code at the front of the while loop calculates the midpoint address and calls the comparison function. An interesting difference from the C implementation is the ability to use single test for the result of the comparison function. The TST.L/BGT/BMI series replaces the two C if statements. If both conditional branches are not taken, the result is zero and the midpoint is the return value. The code at labels bs20 and bs30 adjusts the appropriate address, and the code at label bs40 is the while loop conditional test.

Note the use of the unsigned comparison for the addresses. Many systems have code at addresses where bit A31 is a one (a negative number). You should always use unsigned branches for address quantities so that your code can be run on these systems.

Performance To find the average time to perform a search, we measured the total time to search for all records in an array, and divided by the number of records. The following times were obtained on a Macintosh II:

Records	C	Asm	Gain
00001	00050.00	00050.00	00.0%
00002	00058.00	00041.00	29.3%
00004	00075.00	00058.00	22.7%
00008	00089.00	00064.00	28.1%
00016	00109.00	00079.00	27.5%
00032	00129.00	00093.00	27.9%
00064	00152.00	00108.00	28.9%
00128	00176.00	00125.00	29.0%
00256	00200.00	00142.00	29.0%
00512	00225.00	00159.00	29.3%
01024	00250.00	00177.00	29.2%

Graphics Examples

Some of the most intensive optimization of integer programming goes into graphics implementations. With the current emphasis on computer user interfaces based on graphics like the Apple Macintosh, the IBM PC Presentation Manager for OS/2, and the plethora of windowing systems available for UNIX, graphics has become vital. This section will examine routines to draw lines and circles on a bitmapped device. The algorithms for both of these routines are discussed by Foley and Van Dam in *Fundamentals of Interactive Computer Graphics* (Addison-Wesley, 1984).

Bresenham Line Algorithm A rather traditional C implementation for the Bresenham line algorithm is shown below:

```
                                         /***********************************/
line(x1,y1,x2,y2,value)                  /* Bresenham's line algorithm      */
int     x1,y1,x2,y2;                     /* End points of line              */
int     value;                           /* Pixel value to draw             */
{                                        /*                                 */
   int  dx,dy;                           /* Differences x1-x2;y1-y2         */
   int  incr1,incr2;                     /* Temporary increment values      */
   int  d,x,y,xend;                      /* Decision,x,y, ending x value    */
                                         /*---------------------------------*/
   dx = abs(x1 - x2);                    /* Calculate                       */
   dy = abs(y1 -y2);                     /*           differences           */
   d = dy+dy - dx;                       /* 2 * dy-dx                       */
   incr1 = dy + dy;                      /* 2 * dy                          */
   incr2 = (dy-dx) << 1;                 /* 2 * (dy-dx)                     */
                                         /*---------------------------------*/
   if(x1 > x2)                           /* Start with smallest x value     */
   {                                     /*                                 */
     x = x2;                             /* Initial point                   */
     y = y2;                             /*            in line              */
     xend = x1;                          /* Final x value                   */
   }                                     /*                                 */
   else                                  /* Use other set of values         */
   {                                     /*                                 */
     x = x1;                             /* Initial point                   */
     y = y1;                             /*            in line              */
     xend = x2;                          /* Final x value                   */
   }                                     /*                                 */
                                         /*---------------------------------*/
   pixel(x,y,value);                     /* Write initial point             */
   while(x < xend)                       /* Draw the body of the line       */
   {                                     /*                                 */
     x++;                                /* Bump x value                    */
     if(d < 0)                           /* Need to increment y?            */
       d += incr1;                       /* No.                             */
     else                                /* Must increment y.               */
     {                                   /*                                 */
       y++;                              /* Increment y.                    */
       d += incr2;                       /* Increment d as well             */
     }                                   /*                                 */
     pixel(x,y,value);                   /* Write next pixel                */
   }                                     /*                                 */
}                                        ************************************/
```

The routine is called with the two endpoints of the line (x1,y1) and (x2,y2), plus the value to be used for each pixel in the line. This version differs from the one published in the reference in the use of shift operators and additions rather than multiplications.

The pixel() routine is an inherently device-dependent piece of code. Since this routine is part of the innermost loop, it should be in assembly language. The implementation presented here uses the single bit-per-pixel Macintosh screen layout for the graphics device. The origin (0,0) is at the top left of the screen and coordinates increment to the right and down. This layout is common to all Macintosh computers from the 128K Macintosh on. Bits in a byte use the wrong numbering for this layout to be convenient, as we will see. The pixel routine is as follows:

```
        IMPORT   bitmap:DATA       ;    Bitmap pointer
pixel   MOVE.L   bitmap,A0         ;    A0 -> Bitmap array
        MOVEM.L  4(SP),D0/D1/A1    ;    x,y,value
        MOVE.L   D2,-(SP)          ;    Save registers
        LSL.L    #5,D1             ;    Formulate index
        MOVE.L   D0,D2             ;    Copy x value
        LSR.L    #3,D2             ;    Divide by 8
        ADD.L    D2,D1             ;    Byte-level index
        ANDI.L   #7,D0             ;    Calculate bit number
        MOVE.B   #7,D2             ;    And
        SUB.B    D0,D2             ;        invert ordering
        TST.L    A1                ;    Set or clear?
        BEQ.S    clrpix            ;    Clear
        BSET.B   D2,0(A0,D1.L      ;    Set the pixel
        MOVE.L   (SP)+,D2          ;    Restore D2
        RTS                        ;    And return
clrpix  BCLR.B   D2,0(A0,D1.L)     ;    Clear the pixel
        MOVE.L   (SP)+,D2          ;    Restore D2
        RTS                        ;    And return
```

The target bitmap is a 256 x 256 pixel byte array. The number of bytes in a row is 256 ÷ 8, or 64. Increasing Y values are therefore 0, 64, 128, and so on. This routine calculates the address of the byte addressed by the x and y coordinates supplied as arguments as (y*64 + x). The bit number within this byte is (7 - (x&7)). The subtraction from 7 essentially reverses the order of the bits within the byte. Since our target implementation uses only a 2-valued pixel (on or off), we use the BSET and BCLR to turn pixels on or off.

Going to assembly language, we note that the pixel function is called only twice in the entire procedure. Furthermore, of the thirteen variables in the algorithm, only six are used in the single loop in the algorithm. We therefore make the following register assignment:

Register	**Usage**
D2	xend variable
D3	incr1 variable
D4	incr2 variable
D5	d variable
D6	x variable
D7	y variable
A0	used for address of bitmap
A1	value parameter

The pixel() routine has been changed to use a register-based calling sequence. This routine now requires the x coordinate in D0 and the y coordinate in D1. Registers A0 and A1 must also contain the bitmap address and pixel value quantities. The pixel() routine is as follows:

```
pixel   MOVE.L   D2,-(SP)          ;    Save registers
        LSL.L    #5,D1             ;    Formulate index
        MOVE.L   D0,D2             ;    Copy x value
        LSR.L    #3,D2             ;    Divide by 8
        ADD.L    D2,D1             ;    Byte-level index
        ANDI.L   #7,D0             ;    Calculate bit number
        MOVE.B   #7,D2             ;    And
        SUB.B    D0,D2             ;        invert ordering
        TST.L    A1                ;    Set or clear?
        BEQ.S    clrpix            ;    Clear
        BSET.B   D2,0(A0,D1.L)     ;    Set the pixel
        MOVE.L   (SP)+,D2          ;    Restore D2
        RTS                        ;    And return
clrpix  BCLR.B   D2,0(A0,D1.L)     ;    Clear the pixel
        MOVE.L   (SP)+,D2          ;    Restore D2
        RTS                        ;    And return
```

This calling sequence saves a MOVE.L and a MOVEM.L instruction. Although this implementation saves only two instructions out of fifteen, it saves four data references out of seven. This is very important on a CPU with an instruction cache. (The 68020 and later chips have this type of cache.)

The line algorithm has the following assembly language implementation:

```
x1      EQU      8                 ;    Stack offset for x1
y1      EQU      12                ;    Stack offset for y1
x2      EQU      16                ;    Stack offset for x2
y2      EQU      20                ;    Stack offset for y2
value   EQU      24                ;    Stack offset for value
```

```
line:   LINK     A6,#0              ;   Use standard stack frames
        MOVEM.L  D2-D7,-(SP)        ;   Save data registers.
        MOVE.L   bitmap,A0          ;   A0 -> Bitmap data
        MOVE.L   value(A6),A1       ;   A1 = pixel value
        MOVE.L   x1(A6),D0          ;   Calculate
        SUB.L    x2(A6),D0          ;           dx
        BGE.S    li10               ;   already positive
        NEG.L    D0                 ;   Make positive
li10                                ;
        MOVE.L   y1(A6),D1          ;   Calculate
        SUB.L    y2(A6),D1          ;           dy
        BGE.S    li20               ;   Already positive
        NEG.L    D1                 ;   Make positive
li20                                ;
        MOVE.L   D1,D4              ;   incr2 = dy
        SUB.L    D0,D4              ;            - dx
        MOVE.L   D4,D5              ;   d = dy - dx
        ADD.L    D4,D4              ;   incr2       << 1
        ADD.L    D1,D5              ;   d = dy - dx + dy
        MOVE.L   D1,D3              ;   incr1 = dy
        ADD.L    D1,D3              ;              + dy
        MOVE.L   x2(A6),D6          ;   x    = x2;
        CMP.L    x1(A6),D6          ;   if(x1 > x2)
        BGE.S    li30               ;   Not the case
        MOVE.L   y2(A6),D7          ;   y    = y2;
        MOVE.L   x1(A6),D2          ;   xend = x1;
        BRA.S    li40               ;   Branch around next
li30    MOVE.L   x1(A6),D6          ;   x    = x1;
        MOVE.L   y1(A6),D7          ;   y    = y2;
        MOVE.L   x2(A6),D2          ;   xend = x1;
li40                                ;
        MOVE.L   D6,D0              ;   pixel(x,
        MOVE.L   D7,D1              ;         y,value);
        BSR.S    pixel              ;   Do it.
                                    ;
        BRA.S    li80               ;   Use inverted loop
li50    ADDQ.L   #1,D6              ;   x++;
        TST.L    D5                 ;   if(d < 0)
        BGE.S    li60               ;   it isn't
        ADD.L    D3,D5              ;   d += incr1
        BRA.S    li70               ;   Skip next
li60    ADDQ.L   #1,d7              ;   y++;
        ADD.L    D4,D5              ;   d += incr2;
li70    MOVE.L   D6,D0              ;   pixel(x,
        MOVE.L   D7,D1              ;         y,value);
        BSR.S    pixel              ;   Draw the pixel
li80    CMP.L    D6,D2              ;   while(x < xend)
        BGE.S    li50               ;   xend >= x.
        MOVEM.L  (SP)+,D2-D7        ;   Fix what we broke
        UNLK     A6                 ;   Pop frame
        RTS                         ;   Back to caller
```

The equate statements at the top of the listing define offsets from A6 for the parameters. Note that the x1, x2, y1, and y2 parameters are not kept in registers, but all other quantities have a register assignment. (A shortage of data registers prevents allocating a register for each parameter.)

The calculation of variables d and incr2 illustrates a useful principle. These variables are initialized with the assignment statements:

```
d = dy+dy - dx;        /* 2 * dy-dx                        */
incr1 = dy + dy;       /* 2 * dy                           */
incr2 = (dy-dx) << 1;  /* 2 * (dy-dx)                      */
```

Note that the difference dy-dx is common to the first and last assignment statements. The code that implements these three statements (just after label li20) reorders the computation such that the difference is taken once rather than twice. This technique is called *common subexpression elimination* and is widely used in optimizing compilers.

The code between labels li50 and li80 implements the while() loop, which draws all but the first pixel. This code and the pixel() routine are the most time-consuming part of this algorithm. Note that none of this code references data memory and that all the data is in registers. There are ten instructions per iteration, plus the thirteen in the pixel() routine. The assembly language implementation is 2.48 times faster than the C implementation.

Bresenham Circle Algorithm The circle algorithm draws a 45° arc (one-eighth of a circle) and uses the fact that the circle is symmetrical to get a complete 360° figure. The C version is:

```
                                      /**********************************/
circle(xc,yc,r,value)                 /* Bresenham circle algorithm     */
int       xc,yc,r;                    /* center / radius                */
int       value;                      /* Value for pixels               */
{                                     /*--------------------------------*/
  int     x,y,d;                      /* temporaries                    */
                                      /*                                */
  x = 0;                              /* Initialize                     */
  y = r;                              /*                                */
  d = 3 - r+r;                        /* 3 - 2*radius                   */
  while(x < y)                        /*                                */
  {                                   /*                                */
    circle_points(xc,yc,x,y,value);   /* Draw 8-way points              */
    if(d < 0)                         /* Don't move y value             */
      d += (x<<2) + 6;                /* New decision value             */
    else                              /*                                */
    {                                 /* Must move y value            */
      d += ((x-y)<<2) + 10;           /* New value for decision       */
      y--;                            /* Decrement y                  */
    }                                 /*                              */
    x++;                              /* Bump x for next iteration    */
  }                                   /*                              */
  if(x == y)                          /*                              */
    circle_points(xc,yc,x,y,value);   /* 8-way point again            */
}                                     /********************************/
```

Again, we use addition and shifting in place of the multiplication in the original algorithm. This function draws a circle centered at point (xc,yc) with radius r. The value parameter is the pixel value for all the points in the circle.

As noted above, this function calculates only one-eighth of the points on the circle. The function circle_points() draws eight symmetric points to complete the circle. This function is as follows:

```
                                      /********************************/
circle_points(xc,yc,x,y,value)        /* 8-way symmetry for circles   */
int     xc,yc;                        /* Center coordinates           */
int     x,y;                          /* Coordinates to replicate     */
int     value;                        /* Pixel value                  */
{                                     /*------------------------------*/
   pixel(xc+x,yc+y,value);            /* 8-                           */
   pixel(xc+y,yc+x,value);            /*                              */
   pixel(xc+y,yc-x,value);            /*                              */
   pixel(xc+x,yc-y,value);            /*       way                    */
   pixel(xc-x,yc-y,value);            /*                              */
   pixel(xc-y,yc-x,value);            /*                              */
   pixel(xc-y,yc+x,value);            /*              symmetry        */
   pixel(xc-x,yc+y,value);            /*                              */
}                                     /*                              */
                                      /********************************/
```

The pixel routine used here is the same as the one presented for the C version of the line algorithm. Converting to assembly language, we used the same pixel code as the line algorithm with the register-based calling sequence. We also converted the circle_points() routine to use register parameters. The register conventions are as follows:

Register	Usage
D3	d variable
D4	xc variable
D5	yc variable
D6	x variable
D7	y variable
A0	used for address of bitmap
A1	value parameter

Note that many of the registers are the same as that for the line routine. In commercial programming, consistency is important because the author of a program is not the person who maintains the program. Readability is also very important in commercial software. The circle algorithm is implemented as follows:

```
circle:  LINK     A6,#0
         MOVEM.L  D2-D7,-(SP)        ;    Save registers
         MOVEM.L  8(A6),D4/D5/D7/A1  ;    Load arguments (y=r)
         CLR.L    D6                 ;    x = 0;
         MOVE.L   D7,D3              ;    d = r
         ADD.L    D3,D3              ;          + r
         NEG.L    D3                 ;    - r+r
         ADDQ.L   #3,D3              ;    3 - r+r
                                     ;
         BRA.S    ci40               ;    Top of while()
ci10:    BSR.S    circle_points      ;    Do the circle stuff
         TST.L    D3                 ;    if(d < 0)
         BGT.S    ci20               ;    It isn't
         MOVE.L   D6,D0              ;    Temp.
         ASL.L    #2,D0              ;    x << 2
         ADD.L    D0,D3              ;    d += x << 2
         ADDQ.L   #6,D3              ;              + 6
         BRA.S    ci30               ;    Skip the next part
ci20:    MOVE.L   D6,D0              ;    x
         SUB.L    D7,D0              ;      - y
         ASL.L    #2,D0              ;         << 2
         ADD.L    D0,D3              ;    d += (x-y)<<2
         ADDQ.L   #8,D3              ;                +
         ADDQ.L   #2,D3              ;                  10
         SUBQ.L   #1,D7              ;    y--;
ci30:    ADDQ.L   #1,D6              ;    x++;
ci40:    CMP.L    D6,D7              ;    while(x < y)
         BGT.S    ci10               ;    Restart loop
         CMP.L    D6,D7              ;    if(x == y)
         BNE.S    ci50                ;    It isn't
         BSR.S    circle_points       ;    Do the 8-way stuff
ci50:    MOVEM.L  (SP)+,D2-D7         ;    Unsave registers
         UNLK     A6                  ;    Undo stack frame
         RTS                          ;    And return
```

The critical area of code lies between labels ci10 and ci40, which is the body of the while loop. Other than instruction fetches, there are no memory references inside this loop. Another interesting quirk is the use of two ADDQ instructions to add 10 to register D3 (near label ci20). This technique is 2 bytes shorter than an ADDI.L instruction. Assigning a register to contain the value 10 would be better, but there are no free registers.

The circle_points() routine simply uses the register setup for the circle() routine. This code is as follows:

```
         IMPORT   pixel               ;    import the pixel routine
circle_points:                        ;
         MOVE.L   D4,D0               ;1   pixel(xc
         ADD.L    D6,D0               ;            +x,
         MOVE.L   D5,D1               ;               yc
         ADD.L    D7,D1               ;                 +y);
         BSR      pixel               ;    Do the call
         MOVE.L   D4,D0               ;2   pixel(xc
         ADD.L    D7,D0               ;            +y,
         MOVE.L   D5,D1               ;               yc
         ADD.L    D6,D1               ;                 +x);
         BSR      pixel               ;    Do the call
         MOVE.L   D4,D0               ;3   pixel(xc
         ADD.L    D7,D0               ;            +y,
         MOVE.L   D5,D1               ;               yc
         SUB.L    D6,D1               ;                 -x);
         BSR      pixel               ;    Do the call
         MOVE.L   D4,D0               ;4   pixel(xc
         ADD.L    D6,D0               ;            +x,
         MOVE.L   D5,D1               ;               yc
         SUB.L    D7,D1               ;                 -y);
         BSR      pixel               ;    Do the call
         MOVE.L   D4,D0               ;5   pixel(xc
         SUB.L    D6,D0               ;            -x,
         MOVE.L   D5,D1               ;               yc
         SUB.L    D7,D1               ;                 -y);
         BSR      pixel               ;    Do the call
         MOVE.L   D4,D0               ;6   pixel(xc
         SUB.L    D7,D0               ;            -y,
         MOVE.L   D5,D1               ;               yc
         SUB.L    D6,D1               ;                 -x);
         BSR      pixel               ;    Do the call
```

```
MOVE.L   D4,D0          ;7  pixel(xc
SUB.L    D7,D0          ;           -y,
MOVE.L   D5,D1          ;              yc
ADD.L    D6,D1          ;                +x);
BSR      pixel          ;   Do the call
MOVE.L   D4,D0          ;8  pixel(xc
SUB.L    D6,D0          ;           -x,
MOVE.L   D5,D1          ;              yc
ADD.L    D7,D1          ;                +y);
BSR      pixel          ;   Do the call
RTS                     ;   Return
```

This routine also contains no data memory references. The pixel() routine is the same as the one used by the line() algorithm described above. The assembly language version of circle() is 2.16 times faster than the C version.

Floating Point Examples

This section focuses on optimizing performance of floating point algorithms by hand coding them in assembly language. We will optimize programs for the 68881 and 68882 floating point coprocessors. Examples in this section will not run on a 68000 or 68010 system.

There are two sample programs presented here: a brute-force algorithm for finding roots of non-linear equations and a subroutine for solution of simultaneous linear equations. Both are designed to be called as C functions. We use double-precision parameters rather than extended precision since some versions of C do not support the extended precision data type (notably pcc on UNIX).

Solving an equation by the bisection method The bisection method is one of the simplest and most direct ways of finding a root of an equation of a single variable. This solution requires two endpoints, x1 and x2, one where f(x1) is greater than zero and f(x2) is less than zero. By taking the midpoint and adjusting one of the ends depending on the sign of the function at the midpoint, you can find the point at which the equation crosses zero to an arbitrary precision (bounded, of course, by what your floating point support can give you). This method is described in most numerical analysis textbooks.

The implementation here is called bisect() and is similar to the bsearch() function described above. You supply a pointer to the function, the two endpoints, and an error quantity, and bisect() returns a value for the root of the equation within the error boundary. The C calling conventions for this function are

```
double  bisect()                        /* Double precision return value  */
double  rval;                           /* A place to put it              */
double (*f)();                          /*-> Function                     */
double  a,b;                            /*= end points                    */
double  eps;                            /*= error value                   */

rval = bisect(f,a,b,eps);               /* Find root                      */
```

The f argument is the address of a function, which takes one double-precision value and returns the value of the function at that point.

The C implementation for the bisect() function is as follows:

```
double bisect(f,a,b,eps)                / *********************************/
double      (*f)(double a);             /* function to evaluate           */
double      a,b;                        /* Endpoints                      */
double      eps;                        /* Error criteria                 */
{                                       /*--------------------------------*/
        double    fabs(double a);       /* Floating absolute value        */
        double    fa,fb,fm;             /* Temporary function values      */
        double    mid;                  /* Midpoint value                 */
                                        /*--------------------------------*/
        fa = (*f)(a);                   /* Value at one endpoint          */
        if(fa * (fb=(*f)(b)) > 0)       /* Look for sign change           */
         return 0;                      /* No solution in interval        */
                                        /*--------------------------------*/
        while(fabs(a-b) >= eps)         /* Until get small enough error   */
        {                               /*                                */
         mid = (a+b)/2.0;               /* Calculate midpoint             */
         fm = (*f)(mid);                /* Function at midpoint           */
         if((fm * fa) < 0)              /* Sign change?                   */
         {                              /*                                */
          b = mid;                      /* Sign change here? move b.      */
          fb = fm;                      /* Move function value also       */
         }                              /*                                */
         else                           /* Sign change from mid to b.     */
         {                              /*                                */
          a = mid;                      /* Move a instead.                */
          fa = fm;                      /* Move function value also       */
         }                              /*                                */
        }                               /*                                */
        return mid;                     /* Return calculated value        */
                                        / *********************************/
```

Note the check at the beginning to see that the function in fact crosses zero in the specified interval. The while loop terminates when the value of the difference between the two endpoints is less than the user-supplied error value. By checking the sign of the value of the function at the midpoint against the value of the function at one of the endpoints, you can determine whether the zero-crossing point lies between a and the midpoint, or b and the midpoint. Once you determine where the crossing occurs, move the other endpoint, and you will increase your accuracy by a factor of two. You can then iterate this process until the interval is within the error boundary.

To translate this function to assembly language, we note that there are six floating point values, but some of them are not required for the whole function. In particular, the expression fabs(a-b) and the value of the function at the midpoint (variable fm) are used only briefly. Since there are no intervening procedure calls, you can use the registers that are destroyed by procedure calls (FP0 and FP1) for these variables. The register assignment for this function becomes:

Register	Usage
A4	f parameter to bisect
FP0	fm (value of function at midpoint)
FP1	Temporary
FP2	a (lower end of the selected interval)
FP3	b (upper end of the selected interval)
FP4	eps (error value)
FP5	fa (value of function at point a)
FP6	fb (value of function at point b)
FP7	mid (midpoint between a and b)

The assembly language implementation is shown below. The first part of the function loads the registers and performs the sanity check (that is, is there a zero crossing within the interval?).

```
bisect  LINK     A6,#0              ; Preserve stack frames
        MOVE.L   A4,-(SP)           ; Save integer registers
        FMOVEM   FP2-FP7,-(SP)      ; Save Floating registers
        MOVE.L   8(A6),A4           ; Function pointer
        FMOVEM   12(A6),FP2-FP4     ; Load up the arguments
        FMOVE.X  FP2,-(SP)          ; Calculate f(a)
        JSR      (A4)               ; Call f
        FMOVE    FP0,FP5            ; Save in fa
        LEA      12(SP),SP          ; Pop argument
        FMOVE.X  FP3,-(SP)          ; Now get
```

```
JSR      (A4)               ; f(b)
FMOVE    FP0,FP6            ; Save in fb
LEA      12(SP),SP          ; Pop argument
FMUL     FP5,FP0            ; fa * fb...
FBNGE    bi10               ; Branch if negative
FMOVEM   (SP)+,FP2-FP7      ; Pop floating regs
MOVE.L   (SP)+,A4           ; Pop integer regs
FMOVE.B  #0,FP0             ; Return zero
UNLK     A6                 ; Unframe
RTS                         ;
```

The register assignment means that you can use a single FMOVEM instruction to load all the floating point parameters into floating point registers. This section does a fair job of mixing coprocessor and main processor instructions.

The real place to optimize is the latter part of the program. The first part is executed only once per call; the next section is executed many times before the function returns the desired value to the caller.

```
bi10  BRA.S    bi40          ;    While loop begins
bi20  FMOVE    FP2,FP7       ;    mid =
      FADD     FP3,FP7       ;          a + b
      FDIV.B   #2,FP7        ;                /2
      FMOVE.X  FP7,-(SP)     ;    Push argument
      JSR      (A4)          ;    Call f
      FMOVE    FP0,FP1       ;    Save result (temp)
      FMUL     FP5,FP1       ;    f(a) * f(mid)
      LEA      12(SP),SP     ;    Pop argument to f()
      FBNLE    bi30          ;    It's positive
      FMOVE    FP7,FP3       ;    b = mid
      FMOVE    FP0,FP6       ;    fb = fmid
      BRA.S    bi40          ;    Now test termination
bi30  FMOVE    FP7,FP2       ;    a = mid
      FMOVE    FP0,FP5       ;    fa = fmid
                             ;
bi40  FMOVE    FP2,FP1       ;    Calculate
      FSUB     FP3,FP1       ;              a - b
      FABS     FP1           ;    take absolute value
      FCMP     FP4,FP1       ;    eps : abs(a-b)
      FBGT     bi20          ;    If (a-b) > eps, loop
      FMOVE    FP7,FP0       ;    Set return value
      FMOVEM   (SP)+,FP2-FP7 ;    Pop floating registers
      MOVE.L   (SP)+,A4      ;    Pop procedure pointer
      UNLK     A6            ;    Unframe
      RTS                    ;    Return to caller
```

Note the enormous number of source-to-destination register conflicts in this section. Register FP1 is the only temporary register available in many places, and thus is used in many of the instructions. There is doubtless room for improvement. However, in clocking this routine on a 68020/68881 combination, the assembly language version is 44.5 percent faster than the C version. On a 68030/68882 combination, this improvement is less (31.7 percent), but still significant.

Solution of Simultaneous Linear Equations

Solving a set of simultaneous linear equations consumes large quantities of CPU time. There are numerous techniques for handling this application, but all are CPU-intensive. The technique used here is an algorithm called Gaussian elimination, which is described in most numerical analysis textbooks. There are many variations on the Gaussian elimination theme; the implementation presented here is a C translation of a FORTRAN program published by Dorn and McCracken in *Numerical Methods with FORTRAN IV* (John Wiley, 1972). There are many better techniques, but our purpose is to illustrate some guidelines for optimizing numerical programs. This example illustrates some of the hidden computations to consider when writing numerical software.

C Implementation The C version of this software is in a subroutine called gauss(), and has the following C calling conventions:

```
double        a[NMAX,NMAX+1];     /* Array of coefficients        */
double        x[NMAX];            /* Solution vector              */
int           n;                  /* = number of equations        */

gauss(a,x,n);                     /* Solve the equations          */
```

The array dimensions are statically declared to simplify indexing in the gauss() routine. This subroutine has the following C implementation:

```
void gauss(a,x,n)                        /* gaussian elimination            */
double    a[NMAX][NMAX+1];               /*= input matrix                   */
double    x[NMAX];                       /*= solution vector                */
int       n;                             /*= Number of unknowns             */
{                                        /----------------------------------*/
   int    i,j,k,l;                       /* Temporary indices               */
   double       temp;                    /* Temporary double value          */
                                         /*---------------------------------*/
                                         /* Triangulate matrix 'a'          */
   for(k = 0; k < n-1; k++)              /* Begin partial pivot operation   */
   {                                     /*                                 */
    l = k;                               /* Initial pivot row guess         */
                                         /*---------------------------------*/
    for(i = k+1; i < n ; i++)            /* Find largest column item        */
     if(fabs(a[i][k]) >                  /*                                 */
        fabs(a[l][k])   ) l = i;         /* scan all items in column        */
                                         /*---------------------------------*/
    if(l != k)                           /* Need to interchange rows?       */
    {                                    /* Yes.                            */
     for(j = k; j <= n; j++)             /* Exchange the row.               */
     {                                   /*                                 */
        temp = a[k][j];                  /* Do                              */
        a[k][j] = a[l][j];               /*    the                          */
        a[l][j] = temp;                  /*          exchange               */
     }                                   /*                                 */
    }                                    /*                                 */
    for(i = k+1; i < n; i++)             /* Loop on rows                    */
    {                                    /*                                 */
     temp = a[i][k]/a[k][k];             /* Divide by diagonal element      */
     for(j=k+1; j <= n; j++)             /* Loop on columns                 */
       a[i][j] -= (temp*a[k][j]);        /* Adjust entire row               */
    }                                    /*                                 */
   }                                     /* for(k=0; ...                    */
                                         /*---------------------------------*/
                                         /* Back substitution starts here   */
    x[n-1] = a[n-1][n] / a[n-1][n-1];    /* Solve last unknown first        */
    for(i = n-2; i >= 0; i--)            /* Until get to the end            */
    {                                    /*                                 */
     temp = 0;                           /* Initialize sum of row           */
     for(j = i+1; j < n; j++)            /* Loop through remaining elts     */
        temp += a[i][j] * x[j];          /* Calculate RHS                   */
     x[i] = (a[i][n]-temp)/a[i][i];      /* solve for next unknown          */
    }                                    /*                                 */
}                                        /**********************************/
```

The array a[NMAX][NMAX+1] contains the coefficients of a series of n linear equations of n variables. These equations have the general form:

$$
\begin{array}{l}
a_{0,0}X_0 + a_{0,1}X_1 + \ldots\ a_{0,n-1}X_{n-1} = a_{0,n} \\
a_{1,0}X_0 + a_{1,1}X_1 + \ldots\ a_{1,n-1}X_{n-1} = a_{1,n} \\
\qquad\qquad\qquad \vdots \\
a_{n-1,0}X_0 + a_{n-1,1}X_1 + \ldots\ a_{n-1,n-1}X_{n-1} = a_{n-1,n}
\end{array}
$$

where the subscripts above correspond to subscripts in the C array structure. The first part of the function reduces the matrix a to a triangular form, such that the last row of the matrix has two non-zero elements ($a_{n-1,n-1}$ and $a_{n-1,n}$). Row n-2 has elements ($a_{n-2,n-2}$, $a_{n-2,n-1}$, and $a_{n-2,n}$ and so on), which are non-zero. This arrangement of the matrix is called *triangular form.* Note that you can solve the system of equations by traversing the matrix in reverse order, solving first for x_{n-1} in row n-1, x_{n-2} in row n-2, and so on. This process is called *back substitution* and is performed in the latter part of the listing.

Assembly Language Implementation Converting this rather small program to assembly language reveals how much work the compiler does, especially in matrix operations. We will take the implementation a step at at time, translating small sections of C code into assembler.

We make the following register assignments:

Register	Usage
A0	Base address of coefficient matrix a
A1	Base address of solution vector x
A2-A4	Temporary address quantities
D0	Value of local variable i
D1	Local variable j
D2	Local variable k
D3	Local variable l
D4	Number of equations (parameter n)
D5,D6	Temporary index quantities
D7	NMAX+1 constant

The first part of the subroutine is

```
void gauss(a,x,n)                        /* gaussian elimination          */
double   a[NMAX][NMAX+1];                /*= input matrix                 */
double   x[NMAX];                        /*= solution vector              */
int      n;                              /*= Number of unknowns           */
{                                        /*-------------------------------*/
   int   i,j,k,l;                        /* Temporary indices             */
   double      temp;                     /* Temporary double value        */
                                         /*-------------------------------*/
                                         /* Triangulate matrix 'a'        */
   for(k = 0; k < n-1; k++)              /* Begin partial pivot operation */
   {                                     /*                               */
    l = k;                               /* Initial pivot row guess       */
                                         /*-------------------------------*/
```

The entry part of the procedure entails setting up some registers and initializing the first major for() loop. This code translates as:

```
NMAX    EQU       50                   ;    Largest row number
gauss   LINK      A6,#0                ;    Preserve stack frames
        MOVEM.L   D2-D7/A2-A4,-(SP)    ;    Save integer registers
        FMOVEM.X  FP2/FP3,-(SP)        ;    Save Floating registers
        MOVEM.L   8(A6),A0/A1/A2       ;    Arguments
        MOVE.L    A2,D4                ;    (n belongs in data register)
        MOVE.L    #NMAX+1,D7           ;    For indexing
                                       ;
        CLR.L     D2                   ;    k = 0
ga10    MOVE.L    D4,D5                ;    Calculate n
        SUBQ.L    #1,D5                ;                - 1
        CMP.L D2,D5                    ;    k : n-1
        BLE ga100                      ;    Loop exceeded
                                       ;
        MOVE.L    D2,D3                ;    l = k;
```

The parameters a, x, and n are placed in registers A0,A1, and D4, respectively. Register D7 receives the constant quantity NMAX+1 for use in later indexing. The for() loop initialization sets the local variable k (D2) to zero and terminates the loop whenever k exceeds the expression n-1.

The next section is a bit more involved. This part of the function searches the kth column and finds the largest number, placing the index in local variable l. The C code is

```
    for(i = k+1; i < n ; i++)        /* Find largest column item      */
     if(fabs(a[i][k]) >              /*                               */
      fabs(a[l][k])     ) l = i;     /* scan all items in column      */
```

The assembly language translation is

```
         MOVE.L   D2,D0                  ;    for(i=k
         ADDQ.L   #1,D0                  ;           +1;
         MOVE.L   D0,D5                  ;    Compute
         MULU.L   D7,D5                  ;           a[i][k]
         ADD.L    D2,D5                  ;                   index
         MOVE.L   D3,D6                  ;    Compute
         MULU.L   D7,D6                  ;           a[l][k]
         ADD.L    D2,D6                  ;                   index
                                         ;
ga20     CMP.L    D0,D4                  ;              i < n;
         BLE.S    ga40                   ;    (i >= n) -- terminate
                                         ;
         FMOVE.D  0(A0,D5.L*8),FP0       ;    a[i][k]
         FABS.X   FP0                    ;    take fabs(a[i][k])
                                         ;
         FMOVE.D  0(A0,D6.L*8),FP1       ;    a[l][k]
         FABS.X   FP1                    ;    take fabs(a[i][k])
                                         ;
         FCMP.X   FP0,FP1                ;    a[i][k] : a[l][k]
         FBNLE    ga30                   ;    Don't move l
         MOVE.L   D0,D3                  ;    l = i
         MOVE.L   D5,D6                  ;    Copy index as well
ga30     ADD.L    D7,D5                  ;    Bump up index
         ADDQ.L   #1,D0                  ;                   i++)
         BRA.S    ga20                   ;    Back to loop top
```

The index i increments by 1. The variable l may or may not be set to i at some point in the comparison. We use registers D5 and D6 as the indices for a[i][k] and a[l][k]. These change value along with variables i and l. These index quantities, called induction variables, eliminate the need for expensive multiply operations for computing array indices. (The compiler generates the MOVE/MUL/ADD sequence above the label ga20 for each array reference.)

Note the use of the 68020 scaling factor on the two FMOVE instructions. Since a double-precision floating point quantity is 64 bits long, you can leave the values of the C index variables i, j, l, and k in registers and use this addressing mode directly. This makes the assembly language a bit easier to debug. By incrementing the registers by 8 instead of by 1, a bit of execution efficiency would be gained. This technique is left as an exercise.

The next section of this procedure exchanges rows l and k if these two variables are not equal. The C code for this operation is

```
if(l != k)                          /* Need to interchange rows?      */
{                                   /* Yes.                           */
 for(j = k; j <= n; j++)            /* Exchange the row.              */
 {                                  /*                                */
  temp = a[k][j];                   /* Do                             */
  a[k][j] = a[l][j];                /*    the                         */
  a[l][j] = temp;                   /*          exchange              */
 }                                  /*                                */
}                                   /*                                */
```

Note the expensive array indexing. The compiler generates this copy loop using floating point instructions, although integer copy instructions would be faster. Translating this loop to assembly language yields:

```
ga40                                    ;   Here when loop terminates
        MOVE.L   D2,D5                  ;   Compute
        MULU.L   D7,D5                  ;         a[k]
        ADD.L    D2,D5                  ;               [k]
        LEA      0(A0,D5.L*8),A2        ;   A2 -> a[k][k]
        MOVE.L   A2,A4                  ;   Save for later
        CMP.L    D3,D2                  ;   if(l != k)
        BEQ.S    ga55                   ;   Equal, don't switch
        MOVE.L   D3,D5                  ;   Compute
        MULU.L   D7,D5                  ;         a[l]
        ADD.L    D2,D5                  ;               [k]
        LEA      0(A0,D5.L*8),A3        ;   A3 -> a[l][k]
        MOVE.L   D4,D5                  ;   D5 = n
        SUB.L    D2,D5                  ;         - k (DBRA Count)
ga50    MOVE.L   (A2),D6                ;   Exchange
        MOVE.L   (A3),(A2)+             ;          a
        MOVE.L   D6,(A3)+               ;             long
        MOVE.L   (A2),D6                ;   Exchange
        MOVE.L   (A3),(A2)+             ;          a
        MOVE.L   D6,(A3)+               ;             long
        DBRA     D5,ga50                ;   Repeat as needed
ga55                                    ;
```

Note that the C loop begins with a[k][k] and ends with a[k][n]. We compute the addresses of a[k][k] and a[l][k], and a DBRA count. The exchange operation uses longword instructions with the autoincrement addressing mode.

The next section of code is the matrix triangularization code. This section has the following C implementation:

```
    for(i = k+1; i < n; i++)        /* Loop on rows                    */
    {                               /*                                 */
      temp = a[i][k]/a[k][k];       /* Divide by diagonal element      */
      for(j=k+1; j <= n; j++)       /* Loop on columns                 */
        a[i][j] -= (temp*a[k][j]);  /* Adjust entire row               */
    }                               /*                                 */
  }                                 /* for(k=0; ...                    */
```

Note that the address of the matrix rows a[i][..] and a[k][..] are the only two used in this for() loop. By calculating these once in the outer loop, you can realize a substantial execution improvement. Another possible savings stems from the fact that array element a[k][k] is used repeatedly, but the value of k remains constant through this loop. The assembly language translation is

```
         FMOVE.D  (A4),FP3              ;   Fetch a[k][k] once
         MOVE.L   D2,D0                 ;   for(i = k
         ADDQ.L   #1,D0                 ;          +1;
         MOVE.L   D0,D5                 ;   Compute
         MULU.L   D7,D5                 ;         a[i][0]
         LEA      0(A0,D5.L*8),A2       ;                index
         MOVE.L   D2,D6                 ;   Compute
         MULU.L   D7,D6                 ;         a[k][0]
         LEA      0(A0,D6.L*8),A3       ;                index
ga60     CMP.L    D0,D4                 ;              i < n;
         BLE.S    ga90                  ;
         FMOVE.D  0(A2,D2.L*8),FP1      ;   FP1 = a[i][k]
         MOVE.L   D2,D1                 ;   j = k
         ADDQ.L   #1,D1                 ;        + 1
         FDIV.X   FP3,FP1               ;   FP1 = a[i][k] / a[k][k]
ga70                                    ;   for(j=k+1;j <=n; j++)
         CMP.L    D1,D4                 ;   j : n
         BLT.S    ga80                  ;   if LT, inner loop over
         FMOVE.D  0(A3,D1.L*8),FP0      ;   FP0 = a[k][j]
         FMUL.X   FP1,FP0               ;                 * temp
         FMOVE.D  0(A2,D1.L*8),FP2      ;   FP2 = a[i][j]
         FSUB.X   FP0,FP2               ;                - a[k][j]*temp
         FMOVE.D  FP2,0(A2,D1.L*8)      ;   Store in a[i][j]
         ADDQ.L   #1,D1                 ;   j++
         BRA.S    ga70                  ;   Loop again
ga80                                    ;
         ADDQ.L   #1,D0                 ;                    i++)
         LEA      0(A2,D7.L*8),A2       ;   Bump induction pointer also
         BRA.S    ga60                  ;   Repeat the for()
ga90                                    ;   End for(i=k+1;i<n;i++)
                                        ;
         ADDQ.L   #1,D2                 ;   k++ (in for loop)
         BRA      ga10                  ;   Repeat for loop
ga100                                   ;   End triangularization
```

The first part of this code (before label ga60) places the array element a[k][k] in register FP3 and the addresses of elements a[i][0] and a[k][0] in address registers A2 and A3, respectively. Register A2 is a variant on the induction variable theme, allowing use of the values for k and j directly in the FMOVE statements, which access these two rows of the matrix. The outermost for() loop lies between labels ga60 and ga90. The innermost loop lies between labels ga70 and ga80. The two lines between labels ga90 and ga100 are the end of the for loop, which controls the value of variable k. This for loop lies between labels ga10 and ga100.

When execution hits label ga100, the matrix is in triangular form and the equations can be solved by back substitution. The C code for this process is

```
    x[n-1] = a[n-1][n] / a[n-1][n-1];   /* Solve last unknown first       */
    for(i = n-2; i >= 0; i--)           /* Until get to the end           */
    {                                   /*                                */
      temp = 0;                         /* Initialize sum of row          */
      for(j = i+1; j < n; j++)          /* Loop through remaining elts    */
       temp += a[i][j] * x[j];          /* Calculate RHS                  */
      x[i] = (a[i][n]-temp)/a[i][i];    /* solve for next unknown         */
     }                                  /*                                */
}                                       /*********************************/
```

The first line of this section solves for the last unknown (X_{n-1}). The outer for loop (controlled by the variable i) steps backwards though matrix a, solving for one unknown per iteration. The inner loop (controlled by the variable j) adds the terms of the equation until you are ready to solve for the unknown, in the last assignment statement. Once this code has executed, the procedure is complete.

The assembly language version of these two nested loops is

```
        MOVE.L   D4,D5                ;   Calculate
        SUBQ.L   #1,D5                ;            n - 1
        MOVE.L   D5,D6                ;   Compute
        MULU.L   D7,D6                ;         a[n-1][n]
        ADD.L    D4,D6                ;                 index
        FMOVE.D  0(A0,D6.L*8),FP0     ;   FP0 = a[n-1][n]
        SUBQ.L   #1,D6                ;   Now a[n-1][n-1]
        FDIV.D   0(A0,D6.L*8),FP0     ;   FP0 = a[n-1][n]/a[n-1][n-1]
        FMOVE.D  FP0,0(A1,D5.L*8)     ;   x[n-1] =   "
                                      ;
        MOVE.L   D4,D0                ;   i =
        SUBQ.L   #2,D0                ;      n-2;
ga110                                 ;   Top of for()
        TST.L    D0                   ;   Still >= 0?
        BMI.S    ga140                ;   No, we're done.
        FMOVE.B  #0,FP0               ;   temp = 0;
        MOVE.L   D0,D1                ;   for(j=i
        ADDQ.L   #1,D1                ;        +1;
        MOVE.L   D0,D6                ;   Calculate a[i][0]
        MULU.L   D7,D6                ;   For inside loop
```

```
ga120   CMP.L     D1,D4              ;    j < n;
        BLE.S     ga130              ;
        FMOVE.D   0(A1,D1.L*8),FP1   ;    FP1 = x[j]
        MOVE.L    D6,D5              ;    Calculate a[i][j]
        ADD.L     D1,D5              ;                       index
        FMUL.D    0(A0,D5.L*8),FP1   ;    x[j]*a[i][j]
        FADD.X    FP1,FP0            ;    temp += x[j]*a[i][j]
        ADDQ.L    #1,D1              ;    j++
        BRA.S     ga120              ;    To top of for loop
ga130   MOVE.L    D6,D5              ;    Compute a[i][n]
        ADD.L     D4,D5              ;                     index
        FMOVE.D   0(A0,D5.L*8),FP2   ;    FP2 = a[i][n]
        FSUB.X    FP0,FP2            ;                       -temp
        MOVE.L    D6,D5              ;    Compute a[i][i]
        ADD.L     D0,D5              ;                     index
        FDIV.D    0(A0,D5.L*8),FP2   ;    FP2 = (a[i][n]-temp)/a[i][i]
        FMOVE.D   FP2,0(A1,D0.L*8)   ;    put in x[i]
        SUBQ.L    #1,D0              ;    Decrement i
        BRA.S     ga110              ;    Back to loop top
ga140                                ;    End of procedure
        FMOVEM.X  (SP)+,FP2/FP3      ;    Restore floating registers
        MOVEM.L   (SP)+,D2-D7/A2-A4  ;    Restore integer registers
        UNLK      A6                 ;    Unframe
        RTS                          ;    Return to caller
```

Note that in these loops only the row corresponding to the value of i is accessed. Register D6 is used to contain the index for a[i][0]. The address register technique used in the previous section might save some time here. This experiment is also left as an exercise.

Performance Both the C and assembly language versions of the gauss() routine were run on 68020/68881 and 68030/68882 hardware platforms. The test case solved matrices with sizes from 2x2 to 50x50. The 68020 setup showed a 46.3 percent speed improvement (1.86 times faster), while the 68030 showed a 53.5 percent improvement (2.15 times faster).

Miscellaneous

This section contains assembly language code that performs useful functions not available in most high-level languages.

is020() Due to a design flaw in the 68000 and 68010 chips, the extension word for the indexed addressing modes is not verified by the earlier chips. As a result, you can specify a scaled index register and the scaling will be honored on the 68020 and 68030 but ignored on the 68000 and 68010. The routine is020() uses this artifact to return a TRUE/FALSE (1 or 0) indicator of whether the 68020 instruction set is available. This function has the following implementation:

```
is020   MOVE.L   #1,D0                ;    Set index register
        LEA      table,A0             ;-> table
        MOVE.B   0(A0,D0.L*4),D0      ;    get return value
        RTS                           ;    Back to caller
table   DC.B     0,0,0,0              ;    Return value table
        DC.B     1,0,0,0              ;
```

Since this function can be called in either supervisor or user mode, it is usable by applications on any system.

config() A more complete routine figures out the CPU type, FPU type (if any), and whether or not an MMU is present. Each of these three categories is assigned a byte in D0 (the function return register) as follows:

Value (hex)	Interpretation
$00000000	CPU is a 68000
$00000001	CPU is a 68010
$00000002	CPU is a 68020
$00000003	CPU is a 68030
$00000100	68881 FPU present
$00000200	68882 FPU present
$00010000	MMU is present

These values are added together to represent the complete configuration. For example, a 68030 with a 68882 FPU has a value $00010203. The config() routine must run in supervisor mode and be able to access the CPU vector table. The implementation is as follows:

```
IS000   EQU      0                    ;          0 => 68000
IS010   EQU      1                    ;          1 => 68010
IS020   EQU      2                    ;          2 => 68020
IS030   EQU      3                    ;          3 => 68030
                                      ;
IS881   EQU     $100                  ;        100 => 68881 present
IS882   EQU     $200                  ;        200 => 68882 present
                                      ;
MMU     EQU    $10000                 ;     100000 => MMU is present
                                      ;
illvec  EQU    $10                    ;    Illegal instruction vector
linef   EQU    $2C                    ;    Line F vector
```

```
config   MOVE.L   #1,D0                ;    Set index register
         LEA      table,A0             ;    A0 -> table
         MOVE.B   0(A0,D0.L*4),D0      ;    get return value
         BNE.S    got020               ;    It's an 020.
         LEA      illvec,A0            ;    A0-> Illegal vector
         LEA      con20,A1             ;    A1-> Trap handler
         MOVE.L   (A0),-(SP)           ;    Save old vector
         MOVE.L   A1,(A0)              ;    Set new vector
         MOVE.L   SP,A1                ;    Save stack pointer
         MOVE     CCR,D0               ;    See if 010.
con10    MOVE.L   #IS010,D0            ;    It's an 010.
con20    MOVE.L   A1,SP                ;    Restore stack
         MOVE.L   (SP)+,(A0)           ;    Restore vector
         RTS                           ;    D0 = IS000 or IS010
                                       ;
got020   MOVEC    VBR,A0               ;    Get vector base register
         LEA      linef(A0),A0         ;    A0-> Line F trap vector
         MOVE.L   (A0),-(SP)           ;    Save old vector
         LEA      con40,A1             ;    A1-> Trap handler
         MOVE.L   A1,(A0)              ;    Set the vector
         MOVE.L   SP,A1                ;    Remember stack pointer
         FNOP                          ;    Make sure it's idle
         FSAVE    -(SP)                ;    See if got FPU
         CMP.B    #$18,1(SP)           ;    Is it a 68881?
         BEQ.S    con30                ;    No.
         ADD.L    #IS881,D0            ;    Make a 68882
con30    ADD.L    #IS881,D0            ;    or a 68881
con40    MOVE.L   A1,SP                ;    clean up stack
         LEA      con20,A1             ;    Set vector again
         MOVE.L   A1,(A0)              ;    point to exit routine
         MOVE.L   SP,A1                ;    Save stack pointer
         PFLUSHA                       ;    Flush MMU
         ADD.L    #MMU,D0              ;    Set MMU flag
         LEA      con50,A1             ;    Set another vector
         MOVE.L   A1,(A0)              ;    Now see if 030
         MOVE.L   SP,A1                ;    Remember stack
         PSAVE    -(SP)                ;    Only for 851
         BRA.S    con20                ;    Go exit
con50    OR.L     #IS030,D0            ;    We're an 030
         BRA.S    con20                ;    Go exit
table    DC.B     0,0,0,0              ;    CPU value table
         DC.B     2,0,0,0              ;
```

The first part of the code uses the indexing technique to distinguish between the 68000/010 and the 68020/030. For the 68000/010 case, the MOVE CCR instruction is used to make the distinction. On the 68000, this instruction takes an Illegal instruction exception through vector 4. The 68010 will not trap. The 68000 skips the instruction that loads the value #IS010 into D0 and returns zero. Note the use of register A1 to pop the exception frame off the stack. The old contents of the interrupt vector are preserved on the stack and later restored. Since the 68010 never traps, using the Vector Base Register (VBR) is not required.

For the 68020/030 case, we use the VBR and the F-line trap rather than the illegal instruction trap. The first part of this code issues a floating point No Operation (FNOP) instruction. If a coprocessor is present, execution continues normally. Otherwise, the program picks up execution at label con40, which starts the MMU determination. If an FPU is present, the FSAVE instruction places an IDLE state frame on the stack. You can use the size to determine whether the FPU is a 68881 ($18) or a 68882 ($38). Note the clever use of the ADD instructions.

The MMU determination takes place at label con40. The PFLUSHA instruction works on both the 68851 and the 68030. If you get an F-line trap from this instruction, you have no MMU at all (and thus, a 68020 processor). Executing a PSAVE instruction determines whether the MMU is a 68851 (no trap) or 68030 MMU (F-line trap).

Instruction Tracer

As an example of how to deal with exceptions across the 68000 processor family, consider a program that executes instructions in another program one at a time. The trace bit in the status register provides a trace exception after execution of a single instruction. The exception uses vector 9 (address $24). Consider also the problem of how to catch exceptions that occur in the traced instruction. Our example illustrates methods for doing this as well.

But first, the calling sequence. The tracer must be callable from C, with a return value indicating success or error. The registers must be passed as a C structure address. The sequence is

```
long     inregs[18];                     /* 8 D + 8 A + PC + SR          */
long     outregs[18];                    /* Output registers, same order  */
int      rval;                           /* Return value                  */

    rval = tracer(inregs,outregs);
```

The return value is the vector number of the exception. In the normal case, this will be 9 (trace exception). The input registers and output registers are the registers listed above before and after execution of the traced instruction. Using two arrays instead of one enables the software calling tracer() to print out only those registers which have changed. The Status Register (SR) is stored in the low-order word of the last longword in the array. This is more portable than a C structure with one 16-bit quantity in the midst of 32-bit quantities.

The requirement to allow the traced instruction to fault makes the sequence much more difficult. There is no practical way to write this routine to run on both the 68010/020/030 and the 68000; the older chip requires a separate implementation.

68010/020/030 Implementation The newer chips provide a format word that indicates the vector offset when you take an exception. You can therefore have all exceptions routed to the same code and make the vector determination from the format word.

The first part of the tracer() routine initiates the traced instruction. This code is as follows:

```
XD0      EQU     0                         ;    D0 offset
XA0      EQU     8*4                       ;    A0 offset
XSP      EQU     15*4                      ;    Stack pointer offset
XPC      EQU     XSP+4                     ;    Program counter offset
XSR      EQU     XPC+4                     ;    Status register offset
TRACE    EQU     $8000                     ;    Trace bit
RTEINST  EQU     $4E73                     ;    RTE instruction word

tracer:  MOVE.L        4(SP),A0            ;    A0 -> Register area
         LEA           dest,A1             ;    A1 -> Destination reg area
         MOVE.L        8(SP),(A1)          ;    Remember this address
         MOVEM.L       D2-D7/A2-A6,-(SP)   ;    Save registers
         LEA           oldsp,A1            ;    A1 -> SP Save area
         MOVE.L        SP,(A1)             ;    Save this
         MOVE.L        XSP(A0),SP          ;    Switch to his stack
         CLR.W         -(SP)               ;010 Format word
         MOVE.L        XPC(A0),-(SP)       ;    Set PC
         MOVE.L        (SP),A1             ;    A1 -> Target instruction
         MOVE.W        XSR+2(A0),-(SP)     ;    And SR
         ORI.W         #TRACE,(SP)         ;    Set trace bit
         CMP.W         #RTEINST,(A1)       ;    An RTE Instruction?
         BNE.S         NOTRTE              ;    No.
         ORI.W         #TRACE,8(SP)        ;010 Set trace in SR for RTE
NOTRTE   BSR.S         trapsin             ;    Install exception hooks
         MOVEM.L       (A0),D0-D7/A0-A6    ;    Set registers
         RTE                               ;    And trace it.
oldsp:   DC.L          0                   ;    Old stack pointer
dest:    DC.L          0                   ;-> Destination area
```

The definitions at the beginning of this listing define offsets into the register buffer. There are also definitions for the trace bit and the opcode pattern for an RTE instruction.

The code at the label tracer is called using the C calling conventions described above. It is necessary to preserve both the C caller's environment and the environment in the code being traced. The first order of business is to save the destination structure address in a place where it can be fetched using a PC-relative addressing mode. Since MPW globals use register A5 for the global data area, you cannot save the pointer there since the target may destroy A5. The registers that must be preserved in C are saved on the stack.

The next part of the code switches to the target stack and builds a format 0 stack frame. Note the check for an RTE instruction. If you trace an RTE, the RTE will reload the status register. Unless you have set the TRACE bit in the status register image on the stack (or unless it was already set), the TRACE bit will be cleared and the tracing software will lose control forever.

After the RTE check, the exception vectors are set to point to the tracing software exception handler. Once this has been done, the registers are restored to the target values, and an RTE instruction starts execution of the traced program.

When an exception occurs, a routine called except receives control. This is the only routine for all exceptions and appears below:

```
except:  MOVE.L   A0,-(SP)                  ;      Save A0
         MOVE.L   dest,A0                   ;      A0 -> Destination area
         MOVEM.L  D0-D7/A0-A6,(A0)          ;      Save the registers
         MOVE.L   (SP)+,XA0(A0)             ;      Save A0
         MOVE.W   (SP)+,XSR+2(A0)           ;      save status register
         MOVE.L   (SP)+,XPC(A0)             ;      And program counter
         ANDI.W   #TRACE-1,XSR+2(A0)        ;      Clean the trace bit
         MOVE.W   (SP)+,D0                  ;010   Get format word
         MOVE.W   D0,D1                     ;010   Isolate
         LSR.W    #8,D1                     ;010       Format
         LSR.W    #4,D1                     ;010                 bits
         MOVE.B   bytes(D1.W),D1            ;010   Fetch frame size
         ADD.L    D1,SP                     ;      Trim stack
         MOVE.L   SP,XSP(A0)                ;      Save stack
         MOVE.L   oldsp,SP                  ;      Fetch old stack pointer
         BSR.S    trapsout                  ;      Take traps out
         MOVEM.L  (SP)+,D2-D7/A2-A6         ;      Restore registers
         ANDI.L   #$FFF,D0                  ;      Strip to 12 bits
         LSR.W    #2,D0                     ;      Convert to vector number
         RTS                                ;      Done!
bytes:   DC.B     00,00,04,00,00,00,00,00   ; 0-7  Number of bytes
         DC.B     50,12,24,84,00,00,00,00   ; 8-F                to pop
```

This code saves the target registers and reloads the trace code registers. Note the use of the PC-relative data item dest. One of the more interesting problems here is that you need a free address register to perform a MOVEM, but you do not have one. The solution is to park an address register on the stack (in this case, A0), perform the save, and place the stacked value in the proper place in the array.

Next, the exception stack frame must be dismantled. The first 8 bytes are the Status Register (SR), PC, and format word. The SR and PC are placed in the proper offsets in the register array. The format word serves two purposes. The return value is the lower 12 bits of the format word, shifted right by 2 bits. The upper 4 bits of this word indicate how much extra information is contained in the stack frame. Note the two shift instructions in a row. The 68000 shift instructions can only tolerate a 3-bit immediate field (like ADDQ and SUBQ). The number of extra bytes on the stack is contained in the table bytes. In this implementation, we simply discard the information by adding a number from a table to the stack pointer. The stack pointer can then be saved in the destination register array.

The final part of the exception handler switches to the tracing software stack, resets the exception vectors, restores the registers and returns. Note the adjustment to the value in D0 to obtain the vector number rather than the vector offset.

The one remaining piece of code deals with installing and removing the exception vectors. This implementation saves and restores the vector area each time an instruction is traced. Leaving the exception vectors in place is far more efficient, but you run the risk of a catastrophic crash if the tracing software generates an exception. We opted for safety rather than speed. The exception vector code is as follows:

```
NTRAPS   EQU      256                   ;   Number of vectors
trapsin  MOVEC.L  VBR,A2                ;   A2 -> Vector area
         LEA      except,A3             ;   A3 -> New exception handler
         MOVE.W   #NTRAPS-1,D1          ;   D1 = # traps to do
         LEA      traploc,A1            ;   A1 -> Save area
inloop:  MOVE.L   (A2),(A1)+            ;   Save trap
         MOVE.L   A3,(A2)+              ;   Set new trap
         DBRA     D1,inloop             ;   Count down
         RTS                            ;   Go back to beginning

trapsout MOVEC    VBR,A2                ;   A2 -> Vectors
         LEA      traploc,A1            ;   A1 -> Save area
         MOVE.W   #NTRAPS-1,D1          ;   D1 = # traps to do
outloop: MOVE.L   (A1)+,(A2)+           ;   Restore
         DBRA     D1,outloop            ;   and loop
         RTS                            ;   Done.

traploc:                                ; Vector Save Area
```

```
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 00-0F
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 10-1F
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 20-2F
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 30-3F
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 40-4F
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 50-5F
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 60-6F
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 70-7F
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 80-8F
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 90-9F
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; A0-AF
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; B0-BF
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; C0-CF
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; D0-DF
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; E0-EF
        DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; F0-FF
```

Routine trapsin saves the existing vector contents in the table at label traploc. The vectors are then modified to point to routine except (shown above). Routine trapsout undoes this process.

68000 Implementation The 68000 presents a rather different view of the exception mechanism. The format word to identify the source of an exception does not exist. There are only two sizes of stack frames: 6 byte (for normal exceptions) and 14 bytes (for BUSERR and addressing errors). These stack frames appear in Figures 2-34 and 2-35 in Chapter 2. The tracer implementation still uses a single exception routine, however. The trick to accomplish this surfaced while the 68000 was still a novel idea and the origins are uncertain. 68000 chips have 32-bit vectors and program counter registers, but only 24 bits are actually used. If you place information in the upper byte of a vector on a 68000 (or 68010 for that matter), the information will be ignored. Enter a shameless hack. If you place the vector number in the upper byte of the vector contents, the exception handler can derive the vector number by looking at the PC after the exception. The routine presented here uses this technique.

The tracer() routine for the 68000 chip appears below:

```
XD0     EQU     0               ;       D0 offset
XA0     EQU     8*4             ;       A0 offset
XSP     EQU     15*4            ;       Stack pointer offset
XPC     EQU     XSP+4           ;       Program counter offset
XSR     EQU     XPC+4           ;       Status register offset
NTRAPS  EQU     256             ;       Number of vectors
TRACE   EQU     $8000           ;       Trace bit
RTEINST EQU     $4E73           ;       RTE instruction word
```

```
tracer: MOVE.L   4(SP),A0              ;      A0 -> Register area
        LEA      dest,A1               ;      A1 -> Destination reg area
        MOVE.L   8(SP),(A1)            ;      Remember this address
        MOVEM.L  D2-D7/A2-A6,-(SP)     ;      Save registers
        LEA      oldsp,A1              ;      A1 -> SP Save area
        MOVE.L   SP,(A1)               ;      Save this
        MOVE.L   XSP(A0),SP            ;      Switch to his stack
        MOVE.L   XPC(A0),-(SP)         ;      Set PC
        MOVE.L   (SP),A1               ;      A1 -> Target instruction
        MOVE.W   XSR+2(A0),-(SP)       ;      Set SR
        ORI.W    #TRACE,(SP)           ;      Set trace bit
        CMP.W    #RTEINST,(A1)         ;      An RTE Instruction?
        BNE.S    NOTRTE                ;      No.
        ORI.W    #TRACE,6(SP)          ;000   Set trace in SR for RTE
NOTRTE  BSR.S    trapsin               ;      Install exception hooks
        MOVEM.L  (A0),D0-D7/A0-A6      ;      Set registers
        RTE                            ;      And trace it.
oldsp:  DC.L     0                     ;      Old stack pointer
dest:   DC.L     0                     ;->    Destination area
```

The first part of the routine differs only slightly from the preceding version. Since the 68000 has no format word, the ORI.W instruction, which handles tracing RTEs, has a different offset. There is also a missing CLR.W instruction to set up the format word. These are the only two differences in this first part. The exception handler, on the other hand, has undergone many changes:

```
except: MOVE.L   A0,-(SP)              ;      Save the
        MOVE.L   dest,A0               ;      A0 -> Destination area
        MOVEM.L  D0-D7/A0-A6,(A0)      ;      Save the registers
        MOVE.L   (SP)+,XA0(A0)         ;      Save A0
        BSR.S    getpc                 ;000   fetch PC
        NOP                            ;000   Quieten the assembler
getpc   MOVE.L   (SP)+,D0              ;000   D0 = Our PC
        CLR.W    D0                    ;000   Clear low word
        SWAP     D0                    ;000   Isolate
        LSR.W    #$8,D0                ;000          top byte
        CMP.B    #2,D0                 ;000   See if BUSERR
        BLT.S    noadj                 ;000   Don't adjust SP
        CMP.B    #3,D0                 ;000   See if Address error
        BGT.S    noadj                 ;000   Don't adjust
        ADD.L    #8,SP                 ;000   Pop stack stuff
noadj   MOVE.W   (SP)+,XSR+2(A0)       ;      save status register
        MOVE.L   (SP)+,XPC(A0)         ;      And program counter
        ANDI.W   #TRACE-1,XSR+2(A0)    ;      Clean the trace bit
        MOVE.L   SP,XSP(A0)            ;      Save stack
        MOVE.L   oldsp,SP              ;      Fetch old stack pointer
        BSR.S    trapsout              ;      Take traps out
        MOVEM.L  (SP)+,D2-D7/A2-A6     ;      Restore registers
        RTS                            ;      Done!
```

Since there is no format word, you must get the value currently in the PC to obtain the vector number. The BSR to the label getpc places the PC on the stack where it can be retrieved and the high byte isolated. (We use D0 for this purpose.)

The other difference in this area was the stack frame differences. Since there are only two sizes, the check for BUSERR and address error exceptions is done with two compare/branch sequences. The other stack frame difference is that the extra information for the BUSERR/Address error stack frame is pushed after the status register/PC combination rather than before. You must therefore pop this information before rather than after saving the PC and SR.

Since the vectors contain different values and the 68000 has no Vector Base Register, the vector routines had to change as well. The 68000 implementation is as follows:

```
trapsin  SUBA.L   A2,A2                        ;    A2 -> Vector area
         LEA      except,A3                    ;    A3 -> New exception handler
         MOVE.W   #NTRAPS-1,D1                 ;    D1 = # traps to do
         CLR.L    D2                           ;    D2 = counter
         LEA      traploc,A1                   ;    A1 -> Save area
inloop:  MOVE.L   (A2),(A1)+                   ;    Save trap
         MOVE.L   A3,(A2)                      ;    Set new trap
         MOVE.B   D2,(A2)                      ;    Set trap ID
         ADDQ.L   #1,D2                        ;    Fix D2
         ADDQ.L   #4,A2                        ;    Advance A2
         DBRA     D1,inloop                    ;    Count down
         RTS                                   ;    Go back to caller

trapsout SUBA.L   A2,A2                        ;    A2 -> Vectors
         LEA      traploc,A1                   ;    A1 -> Save area
         MOVE.W   #NTRAPS-1,D1                 ;    D1 = # traps to do
outloop: MOVE.L   (A1)+,(A2)+                  ;    Restore
         DBRA     D1,outloop                   ;    and loop
         RTS                                   ;    Done.

traploc:                                       ;
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 00-0F
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 10-1F
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 20-2F
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 30-3F
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 40-4F
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 50-5F
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 60-6F
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 70-7F
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 80-8F
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; 90-9F
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; A0-AF
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; B0-BF
         DC.L     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; C0-CF
DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; D0-DF
DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; E0-EF
DC.L    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; F0-FF
```

Note that SUBA clears an address register. Since the CLR instruction does not allow clearing an address register, this is a short, fast way to perform this operation. Setting the upper byte of the vectors is accomplished by using a separate register. Note that this precludes use of autoincrement addressing on the address register used to address the vectors.

Extensions This version of a tracer is a minimal implementation. There is no provision for tracing user-mode programs, floating point, or a Memory Management Unit. Since the first two of these are rather straightforward extensions, you need pay attention only to the design of the register structure and the order in which the registers are saved and restored. The MMU, while more complicated, is not insurmountable. The problem with an MMU is usually finding the places where a target address is used by the tracer itself. This usually requires a translation process.

Adding these features is left as an exercise. You may find it helpful to look at implementations of debuggers and operating systems for the 68000 family.

Conclusion

The examples in this chapter are guideposts. Assembly language programming on the 68000 or any other chip is best learned by experience. Books like this one can teach you some tricks that can be quite valuable. Information in books is condensed experience. What you read will allow you to design programs which benefit from the experience of others.

There is no substitute for proper design. No amount of optimization can save a poorly designed program. So how do you learn good design? To paraphrase an old joke, good design comes from experience, and experience comes from bad design.

Mistakes are often the best teachers. You don't often read detailed analysis of software design failures in books. (*The Mythical Man-Month* is the exception that proves the rule.) To find out what **does** work, you must often find out what **doesn't** work first. There is no substitute for getting on the machine and testing your design. Computer cost and performance has evolved to a point where you can explore many different design alternatives in a relaperiod of time. The machine will very reliably, very tirelessly tell you what does and doesn't work. Finding out **why** something doesn't work is not nearly as easy, unfortunately.

Another very good way to learn is by reading code written by other people. Unfortunately, large volumes of code are not often published. It is very rare to find a complete source listing for a large program in a bookstore. Dial-up networks like CompuServe and UUCP often have discussion groups where you can find sources to non-trivial programs to look at a variety of styles.

Programming is an adventure. Like all adventures, it has high and low points. You will find a strong correlation between programming and the selective reinforcement theory espoused by behavioral psychologists. Just when you are about to give up, something starts working. The greatest reward is to finish a program and have someone else admire it, or better yet, use it to produce something else of value. That is the reason this book exists. Enjoy.

Appendix A
UNIX Assembly Language

"UNIX is not a system for the meek."
— Charles E. Hughes and J. Michael Moshell

Introduction

This appendix explores writing programs using the UNIX operating system.The brief description of adapting standard Motorola syntax (used throughout the rest of the book) into UNIX assembler addresses the differences in assembly language syntax. The Apple A/UX system is the target system. The example programs from Chapter 6 have been adapted to run under A/UX and are presented here as examples of the conversion process.

Overview

To convert a program from Motorola syntax to the syntax acceptable to *as* (the UNIX assembler), the steps are as follows:

1. Back up the original source file.
2. Change all immediate prefix (#) characters to &.
3. Change all hex prefix ($) characters to 0x. UNIX uses the C language conventions for hex numbers rather than the Motorola syntax. Watch out for decimal constants of the form 0123. The *as* assembler interprets these as octal rather than decimal (consistent with the C language).
4. Change all comment (;) characters to #. If you do not apply steps 2 and 4 in the proper order, you will need to repeat the entire process.

5. Change equate statements as follows:

foo	EQU	value
	becomes	
	set	foo,value

6. All labels must be terminated by the colon (:) character. This is optional in the Motorola syntax and on most other assemblers; it is not optional with UNIX *as*.
7. Change global definitions IMPORT and EXPORT to global. Other assemblers use XREF and XDEF for IMPORT and EXPORT. UNIX uses global for both defining and referencing external symbols.
8. Change data declaration directives as follows:

Motorola	**UNIX**
DC.L	long
DC.W	short
DC.B	byte
DS.L,DS.W,DS.B	space <# of bytes> or comm label,size

   To declare data which is to be referenced by C, use the comm directive. The label should be declared as global.

9. Use the text, data, and bss directives before areas of the program that define code (text), initialized data (data), and uninitialized data (bss). The text and data directives correspond to the MPW CODE and DATA directives, respectively.
10. Convert all instruction mnemonics and register names to lower case. Add a percent (%) character to the beginning of all register names (e.g., D0 becomes %d0). The condition code register (CCR) is %cc for the UNIX assembler.
11. Reverse the operands of all compare instructions.
12. Some UNIX assemblers require that you change the instruction names. For instance, MOVE.L becomes movl. Since the A/UX assembler does not require this, we did not change these names. Some UNIX assemblers do not have a default size, so the instruction MOVE D0,D1 becomes move.w %d0,%d1. This is also true of floating point and MMU instructions.
13. Some UNIX assemblers do not support the different variants of the MOVE instruction. In particular, the MOVEA, MOVEC, and MOVEQ instructions must be coded as move. The assembler automatically figures out the correct instruction based on the operands you supply. The same thing is true of ADDI, ANDI, SUBA, and SUBI.

14. Some UNIX assemblers (including A/UX) do not support all the instructions, despite the manual's claim that the instructions are supported. In particular, A/UX version 1.0 does not support mnemonics for the PMMU instructions, and you must declare them as hex constants.
15. When converting Macintosh programs, ensure that there are no dangling references to A5 on accesses to global data.
16. Advanced assembler features like macros and conditional assembly are not directly supported by the *as* assembler. You can use either the C preprocessor or the m4 macro processor to achieve this functionality.

Program Listings

Following are A/UX source listings for the examples in Chapter 6. These examples have been tested under A/UX using the same test cases as those for MPW.

strcpy. The C string copy routine has two versions: a single-character version, and the multi-character version. The single-character version for A/UX is shown below:

```
        global   strcpy
strcpy: movem.l  4(%sp),&0x300       #   pcc doesn't save these
        move.l   %a0,%d0             #   return value
loop:   move.b   (%a1)+,(%a0)+       #   copy a byte
        bne.s    loop                #   loop
        rts                          #   done
```

The multi-character version is as follows:

```
        global   strcpy
strcpy: movem.l  4(%sp),&0x300       #   mpw doesn't save these
        move.l   %a0,%d0             #   return value
        move.l   %a1,%d1             #   for word / byte
        or.l     %d0,%d1             #   see if
        andi.w   &1,%d1              #         odd
        bne.s    bytes               #   do bytes
        move.l   %d2,-(%sp)          #   save %d2
longs:  move.l   (%a1)+,%d2          #   get next long
        move.l   %d2,%d0             #   copy
        move.l   %d2,%d1                #        twice
        sub.l    &0x01010101,%d0        #    subtract 1 from each byte
        not.l    %d1                    #    1's complement
        and.l    &0x80808080,%d1        #    with 80808080
        and.l    %d1,%d0                #    now and
        bne.s    wbytes                 #    if ne, has a nullbyte
        move.l   %d2,(%a0)+             #    store destination
        bra.s    longs                  #    do some more longs
                                        #
wbytes: move.l   %d2,%d0                #    copy again
        swap     %d0                    #    make it
        lsr.l    &8,%d0                 #        a byte
        lsl.l    &8,%d2                 #    shift to position
        move.b   %d0,(%a0)+             #    store it
        bne.s    wbytes                 #    and around again
        move.l   (%sp)+,%d2             #    restore
        move.l   4(%sp),%d0             #    return value
        move.l   %d0,%a0                #    for pcc
        rts                             #    back to caller
                                        #
bytes:  move.b   (%a1)+,(%a0)+          #    copy a byte
        bne.s    bytes                  #    loop
        move.l   %d0,%a0                #    for pcc
        rts                             #    done
```

strcmp. The C string comparison function has two versions: a single-character version and the multi-character version. The single-character version for A/UX is shown below:

```
        global   strcmp                 #    compare strings
strcmp: movem.l  4(%sp),&0x300          #    fetch a0/a1
cloop:  move.b   (%a0)+,%d0             #    move,
        beq.s    cend                   #    eq means done
        sub.b    (%a1)+,%d0             #    subtract
        beq.s    cloop                  #    keep comparing
        extb.l   %d0                    #    extend d0 to long
        rts                             #    done
cend:   sub.b    (%a1)+,%d0             #    one last time
        extb.l   %d0                    #    extend d0 to long
        rts                             #    done.
```

The multi-character version is as follows:

```
        global   strcmp                 #    compare strings
strcmp: movem.l  4(%sp),&0x300          #    mpw doesn't save these
        move.l   %a0,%d0                #    destination value
        move.l   %a1,%d1                #    for word / byte
```

```
         or.l     %d0,%d1                 #    see if
         andi.w   &1,%d1                  #          odd
         bne.s    cbytes                  #    do bytes
         move.l   %d2,-(%sp)              #    save d2
clongs:  move.l   (%a1)+,%d2              #    get next long
         move.l   %d2,%d0                 #    copy
         move.l   %d2,%d1                 #          twice
         sub.l    &0x01010101,%d0         #    subtract 1 from each byte
         not.l    %d1                     #    1's complement
         and.l    &0x80808080,%d1         #    with 80808080
         and.l    %d1,%d0                 #    now and
         bne.s    cwbytes                 #    if ne, has a nullbyte
         cmp.l    %d2,(%a0)+              #    compare destination
         beq.s    clongs                  #    do some more longs
                                          #
cwbytes: subq.l   &4,%a1                  #    a1 -> back
         move.l   %d2,%d0                 #    copy again
         swap     %d0                     #    make it
         lsr.l    &8,%d0                  #          a byte
         lsl.l    &8,%d2                  #    shift to position
         sub.b    (%a1)+,%d0              #    store it
         bne.s    cwbytes                 #    and around again
         extb.l   %d0                     #    extend d0 to long
         move.l   (%sp)+,%d2              #    restore
         rts                              #    back to caller
                                          #
cbytes:  move.b   (%a0)+,%d0              #    move,
         beq.s    cend                    #    eq means done
         sub.b    (%a1)+,%d0              #    subtract
         beq.s    cbytes                  #    keep comparing
         extb.l   %d0                     #    extend d0 to long
         rts                              #    done
cend:    sub.b    (%a1)+,%d0              #    one last time
         extb.l   %d0                     #    extend d0 to long
         rts                              #    done.
```

_memcpy. The single-byte version of the memory copy routine is as follows:

```
         global   _memcpy
_memcpy: movem.l  4(%sp),&0x300           #    Source / destination
         move.l   12(%sp),%d0             #    Count
         beq.s    cpydone                 #    A wise guy!
cpyloop: move.b   (%a0)+,(%a1)+           #    Move a byte
         subq.l   &1,%d0                  #    Count
         bne.s    cpyloop                 #          down
cpydone: rts                              #    Return to caller
```

The multi-character version is as follows:

```
         set      SIZEL,32
         set      SIZEB,8
         global   _memcpy
_memcpy: movem.l  4(%sp),&0x300         #    Source / destination
         move.l   12(%sp),%d0           #    Count
         move.l   %d0,%d1               #    A scratch register
         cmp.l    %d1,&SIZEB            #    See if too small
         ble.s    cpybyte               #    Just copy bytes.
         or.l     4(%sp),%d1            #    Or in
         or.l     8(%sp),%d1            #          all args
         andi.b   &3,%d1                #    Long aligned?
         bne.s    cpybyte               #    No.
         move.l   &SIZEL,%d1            #    Put size in register
cpyloop: sub.l    %d1,%d0               #    Subtract size 1st
         bmi.s    cpyrest               #    Copy residual
         move.l   (%a0)+,(%a1)+         #    Copy a long
         move.l   (%a0)+,(%a1)+         #    Copy a long
         move.l   (%a0)+,(%a1)+         #    Copy a long
         move.l   (%a0)+,(%a1)+         #    Copy a long
         move.l   (%a0)+,(%a1)+         #    Copy a long
         move.l   (%a0)+,(%a1)+         #    Copy a long
         move.l   (%a0)+,(%a1)+         #    Copy a long
         move.l   (%a0)+,(%a1)+         #    Copy a long
         bra.s    cpyloop               #    and loop
cpyrest: add.l    %d1,%d0               #    Re-add size
         bne.s    cpylong               #    EQ means done.
         rts                            #    back
cpylong: move.l   (%a0)+,(%a1)+         #    Move a long
         subq.l   &4,%d0                #    Count
         bne.s    cpylong               #          down
cpycomp: rts                            #    Return to caller
                                        #
cpybyte: subq.l   &SIZEB,%d0            #    Subtract size 1st
         bmi.s cpyres                   #    Copy residual
         move.b   (%a0)+,(%a1)+         #    Move a byte
         move.b   (%a0)+,(%a1)+         #    Move a byte
         move.b   (%a0)+,(%a1)+         #    Move a byte
         move.b   (%a0)+,(%a1)+         #    Move a byte
         move.b   (%a0)+,(%a1)+         #    Move a byte
         move.b   (%a0)+,(%a1)+         #    Move a byte
         move.b   (%a0)+,(%a1)+         #    Move a byte
         move.b   (%a0)+,(%a1)+         #    Move a byte
         bra.s    cpybyte               #    And around again
cpyres:  addq.l   &SIZEB,%d0            #    Restore
         beq.s    cpycomp               #    Completed.
cpy1:    move.b   (%a0)+,(%a1)+         #    1 / time
         subq.l   &1,%d0                #    And
         bne.s    cpy1                  #          loop
         rts                            #    Finished.
```

_memclr. The single-byte version of the memory clear routine is as follows:

```
         global   _memclr
_memclr: move.l   4(%sp),%a0              #    -> Area to clear
         move.l   8(%sp),%d0              #    =  count
         beq.s    clrdone                 #    A wise guy!
clrloop: clr.b    (%a0)+                  #    Clear a longword
         subq.l   &1,%d0                  #    Count down
         bne.s    clrloop                 #    And Loop
clrdone: rts                              #    Done
```

The multi-character version is as follows:

```
         global   _memclr                 #
_memclr: move.l   4(%sp),%a0              #    -> Area to clear
         move.l   8(%sp),%d0              #    =  count
         move.l   %d0,%d1                 #    Copy count
         cmp.l    %d1,&SIZEB              #    See if too small
         ble.s    clrbyte                 #    Just copy bytes.
         or.l     4(%sp),%d1              #    and OR
         andi.b   &3,%d1                  #    Long aligned?
         bne.s    clrbyte                 #    Clear bytes
         move.l   &SIZEL,%d1              #    Put size in register
clrloop: sub.l    %d1,%d0                 #    Subtract size 1st
         bmi.s    clrrest                 #    Copy residual
         clr.l    (%a0)+                  #    Clear a long
         clr.l    (%a0)+                  #    Clear a long
         clr.l    (%a0)+                  #    Clear a long
         clr.l    (%a0)+                  #    Clear a long
         clr.l    (%a0)+                  #    Clear a long
         clr.l    (%a0)+                  #    Clear a long
         clr.l    (%a0)+                  #    Clear a long
         clr.l    (%a0)+                  #    Clear a long
         bra.s    clrloop                 #    And loop
clrrest: add.l    %d1,%d0                 #    Re-add size
         bne.s    clrlong                 #    EQ means done.
         rts                              #    back
clrlong: clr.l    (%a0)+                  #    Move a long
         subq.l   &4,%d0                  #    Count
         bne.s    clrlong                 #         down
clrcomp: rts                              #    Return to caller
                                          #
clrbyte: subq.l   &SIZEB,%d0              #    Subtract size 1st
         bmi.s    clrres                  #    Copy residual
         clr.b    (%a0)+                  #    Clear a byte
         clr.b    (%a0)+                 #    Clear a byte
         clr.b    (%a0)+                 #    Clear a byte
         clr.b    (%a0)+                 #    Clear a byte
         clr.b    (%a0)+                 #    Clear a byte
         clr.b    (%a0)+                 #    Clear a byte
         clr.b    (%a0)+                 #    Clear a byte
         clr.b    (%a0)+                 #    Clear a byte
         bra.s    clrbyte                #    And around again
clrres:  addq.l   &SIZEB,%d0             #    Restore
         beq.s    clrcomp                #    Completed.
clr1:    clr.b    (%a0)+                 #    1 / time
         subq.l   &1,%d0                 #    And
         bne.s    clr1                   #          loop
         rts                             #    Finished.
```

_memcmp. The single-byte version of the memory compare routine is as follows:

```
         global   _memcmp
_memcmp: movem.l  4(%sp),&0x300          #    Source / destination
         move.l   12(%sp),%d0            #    Count
         beq.s    cmpdone                #    A wise guy!
cmploop: cmp.b    (%a1)+,(%a0)+          #    Move a long
         bne.s    cmpret                 #    Set return value
         subq.l   &1,%d0                 #    Decrement
         bne.s    cmploop                #    And loop
cmpdone: rts                             #    Done.
                                         #
cmpret:  bmi.s    cmpless                #    a < b
         move.l   &1,%d0                 #    > 0# a > b
         rts                             #    Return to caller
cmpless: move.l   &-1,%d0                #    a < b
         rts                             #    Return
```

The multi-character version is as follows:

```
         global   _memcmp
_memcmp: movem.l  4(%sp),&0x300          #    Source / destination
         move.l   12(%sp),%d0            #    Count
         move.l   %d0,%d1                #    A scratch register
         cmp.l    %d1,&SIZEB             #    See if too small
         ble.s    cmpbyte                #    Just compare bytes.
         or.l     4(%sp),%d1              #   Or in
         or.l     8(%sp),%d1              #         all args
         andi.b   &3,%d1                  #   Long aligned?
         bne.s    cmpbyte                 #   No.
         move.l   &SIZEL,%d1              #   Put size in register
cmploop: sub.l    %d1,%d0                 #   Subtract size 1st
         bmi.s    clrrest                 #   Compare residual
         cmp.l    (%a1)+,(%a0)+           #   Compare a long
         bne.s    cmpret                  #   Not equal, quit
         cmp.l    (%a1)+,(%a0)+           #   Compare a long
         bne.s    cmpret                  #   Not equal, quit
         cmp.l    (%a1)+,(%a0)+           #   Compare a long
         bne.s    cmpret                  #   Not equal, quit
         cmp.l    (%a1)+,(%a0)+           #   Compare a long
         bne.s    cmpret                  #   Not equal, quit
         cmp.l    (%a1)+,(%a0)+           #   Compare a long
         bne.s    cmpret                  #   Not equal, quit
         cmp.l    (%a1)+,(%a0)+           #   Compare a long
         bne.s    cmpret                  #   Not equal, quit
         cmp.l    (%a1)+,(%a0)+           #   Compare a long
         bne.s    cmpret                  #   Not equal, quit
         cmp.l    (%a1)+,(%a0)+           #   Compare a long
         bne.s    cmpret                  #   Not equal, quit
         bra.s    cmploop                 #   and loop
cmprest: add.l    %d1,%d0                 #   Re-add size
         bne.s    cmplong                 #   EQ means done.
         rts                              #   back
cmplong: cmp.l    (%a1)+,(%a0)+           #   Move a long
         bne.s    cmpret                  #   Finished.
         subq.l   &4,%d0                  #   Count
         bne.s    cmplong                 #         down
cmpcomp: rts                              #   Return to caller
                                          #
cmpret:  bmi.s    cmpless                 #   a < b
         move.l   &1,%d0                  #   > 0# a > b
         rts                              #   Return to caller
cmpless: move.l   &-1,%d0                 #   a < b
         rts                              #   Return

cmpbyte: subq.l   &SIZEB,%d1              #   Subtract size 1st
         bmi.s    clrres                  #   Compare residual
         cmp.b    (%a1)+,(%a0)+           #   Compare a byte
         bne.s    cmpret                  #   Not equal, quit
         cmp.b    (%a1)+,(%a0)+           #   Compare a byte
         bne.s    cmpret                  #   Not equal, quit
         cmp.b    (%a1)+,(%a0)+          #   Compare a byte
         bne.s    cmpret                 #   Not equal, quit
         cmp.b    (%a1)+,(%a0)+          #   Compare a byte
         bne.s    cmpret                 #   Not equal, quit
         cmp.b    (%a1)+,(%a0)+          #   Compare a byte
         bne.s    cmpret                 #   Not equal, quit
         cmp.b    (%a1)+,(%a0)+          #   Compare a byte
         bne.s    cmpret                 #   Not equal, quit
         cmp.b    (%a1)+,(%a0)+          #   Compare a byte
         bne.s    cmpret                 #   Not equal, quit
         cmp.b    (%a1)+,(%a0)+          #   Compare a byte
         bne.s    cmpret                 #   Not equal, quit
         bra.s    cmpbyte                #   and around again
cmpres:  addq.l   &SIZEB,%d0             #   Restore
         beq.s    cmpcomp                #   Completed.
cmp1:    cmp.b    (%a1)+,(%a0)+          #   1 / time
         bne.s    cmpret                 #   Not equal, quit
         subq.l   &1,%d0                 #   And
         bne.s    cmp1                   #         loop
         rts                             #   Finished.
```

qsort. The A/UX version of the qsort routine is as follows:

```
         global   qsort
qsort:                                   #   Here to sort
         link     %a6,&0                 #   Preserve framing
#        movem.l  %d2-%d7/%a4,-(%sp)     #   Save registers
         movem.l  &0x3f08,-(%sp)         #   Save registers
#        movem.l  8(%a6),%d2-%d3/%d7/%a4 #   4 args
         movem.l  8(%a6),&0x108c         #   4 args
         subq.l   &1,%d3                 #   Subtract 1
         muls.l   %d7,%d3                #   %d3 = size*(nmemb-1)
         add.l    %d2,%d3                #   Add in base to get hi
         bsr.s    _qsort                 #   Call recursive routine
#        movem.l  (%sp)+,%d2-%d7/%a4     #   Pop registers
         movem.l  (%sp)+,&0x10FC         #   Pop registers
         unlk     %a6                    #   Unlink stack frame
         rts                             #   And begone!

         global   _qsort                 #   Qsort internal function
_qsort:                                  #
         cmp.l    %d3,%d2                #   Hi : Lo
         ble      qscomp                 #   LE means done
         move.l   %d2,%d4                #   i = lo
         move.l   %d3,%d5                #   j = hi
         move.l   %d3,%d6                #   pivot = hi
                                         #
```

```
qs10:   cmp.l    %d5,%d4           #   while (i < j)
        ble.s    qs50              #   CMP is inverted, remember...
                                   #
qs20:   bra.s    qs22              #   Skip initial
qs21:   add.l    %d7,%d4           #   i += size;
qs22:   cmp.l    %d5,%d4           #   while ((i < j) &&
        ble.s    qs32              #
#       movem.l  %d4/%d6,-(%sp)    #   Push args
        movem.l  &0x0a00,-(%sp)    #   Push args
        jsr      (%a4)             #   call function
        add.l    &8,%sp            #   Pop args
        tst.l    %d0               #   compar(i,pivot) <= 0))
        ble.s    qs21              #   if not LE, done with while
                                   #
qs30:   bra.s    qs32              #   Skip loop body
qs31:   sub.l    %d7,%d5           #   j -= size;
qs32:   cmp.l    %d5,%d4           #   while ((j > i) &&
        ble.s    qs40              #
#       movem.l  %d5/%d6,-(%sp)    #   Push args
        movem.l  &0x0600,-(%sp)    #   Push args
        jsr      (%a4)             #   call function
        addq.l   &8,%sp            #   Pop args
        tst.l    %d0               #   compar(j,pivot) >= 0))
        bge.s    qs31              #   Done with while
                                   #
qs40:   cmp.l    %d5,%d4           #   if(i < j) out of order
        ble.s    qs10              #   In order, and OK
        move.l   %d4,%a0           #   Set
        move.l   %d5,%a1           #       up
        move.l   %d7,%d0           #         parameters
        bsr.s    exchange          #   exchange(i,j,size);
        bra.s    qs10              #   End of while(i < j)
                                   #
qs50:   move.l   %d4,%a0           #   exchange(i,hi,size);
        move.l   %d3,%a1           #
        move.l   %d7,%d0           #
        bsr.s    exchange          #
                                   #
qs60:   move.l   %d4,%d0           #   i   -
        sub.l    %d2,%d0           #        lo
        move.l   %d3,%d1           #   hi  -
        sub.l    %d4,%d1           #        i
        cmp.l    %d1,%d0           #   if(i-lo < hi - i)
        ble.s    qs70              #   it isn't.
                                   #
```

```
#         movem.l  %d2/%d3/%d4,-(%sp)      #    Need to save lo,hi,i
          movem.l  &0x3800,-(%sp)          #    Need to save lo,hi,i
          move.l   %d4,%d3                 #    replace hi with
          sub.l    %d7,%d3                 #    i - size
          bsr.s    _qsort                  #    qsort(lo,i-size,...);
#         movem.l  (%sp),%d2/%d3/%d4       #    Restore the registers
          movem.l  (%sp),&0x001c           #    Restore the registers
          move.l   %d4,%d2                 #    Replace lo with
          add.l    %d7,%d2                 #    i + size
          bsr.s    _qsort                  #    _qsort(i+size,hi,..);
#         movem.l  (%sp)+,%d2/%d3/%d4      #    Restore original registers
          movem.l  (%sp)+,&0x001c          #    Restore original registers
          rts                              #    And return
                                           #
qs70:                                      #
#         movem.l  %d2/%d3/%d4,-(%sp)      #    Need to save lo,hi,i
          movem.l  &0x3800,-(%sp)          #    Need to save lo,hi,i
          move.l   %d4,%d2                 #    Replace lo with
          add.l    %d7,%d2                 #    i + size
          bsr      _qsort                  #    _qsort(i+size,hi,..);
#         movem.l  (%sp),%d2/%d3/%d4       #    Restore the registers
          movem.l  (%sp),&0x001c           #    Restore the registers
          move.l   %d4,%d3                 #    replace hi with
          sub.l    %d7,%d3                 #    i - size
          bsr      _qsort                  #    qsort(lo,i-size,...);
#         movem.l  (%sp)+,%d2/%d3/%d4      #    Restore original registers
          movem.l  (%sp)+,&0x001c          #    Restore original registers
qscomp:   rts                              #    And return
          set      SIZEL,32                #    Size for long exchanges
          set      SIZEB,8                 #    Size for byte exchanges
exchange:                                  #    Here to exchange
          move.l   %d2,-(%sp)              #    Save a temporary
          cmp.l    %d0,&SIZEB              #    See if too small
          ble.s    excbyte                 #    Just copy bytes.
          move.l   %a0,%d2                 #    See
          or.l     %d0,%d2                 #         if
          move.l   %a1,%d1                 #           Longs
          or.l     %d1,%d2                 #                OK
          andi.b   &3,%d2                  #    Long aligned?
          bne.s    excbyte                 #    No.
          move.l   &SIZEL,%d1              #    Put size in register
excloop:  sub.l    %d1,%d0                 #    Subtract size 1st
          bmi.s    excrest                 #    Copy residual
                                           #0
          move.l   (%a0),%d2               #    Fetch
          move.l   (%a1),(%a0)+            #          and
          move.l   %d2,(%a1)+              #             exchange
                                           #1
```

```
         move.l   (%a0),%d2                #    Fetch,
         move.l   (%a1),(%a0)+             #          and
         move.l   %d2,(%a1)+               #             exchange
                                           #2
         move.l   (%a0),%d2                #    Fetch,
         move.l   (%a1),(%a0)+             #          and
         move.l   %d2,(%a1)+               #             exchange
                                           #3
         move.l   (%a0),%d2                #    Fetch,
         move.l   (%a1),(%a0)+             #          and
         move.l   %d2,(%a1)+               #             exchange
                                           #4
         move.l   (%a0),%d2                #    Fetch,
         move.l   (%a1),(%a0)+             #          and
         move.l   %d2,(%a1)+               #             exchange
                                           #5
         move.l   (%a0),%d2                #    Fetch,
         move.l   (%a1),(%a0)+             #          and
         move.l   %d2,(%a1)+               #             exchange
                                           #6
         move.l   (%a0),%d2                #    Fetch,
         move.l   (%a1),(%a0)+             #          and
         move.l   %d2,(%a1)+               #             exchange
                                           #7
         move.l   (%a0),%d2                #    Fetch,
         move.l   (%a1),(%a0)+             #          and
         move.l   %d2,(%a1)+               #             exchange
                                           #
         bra.s    excloop                  #    And loop
excrest: add.l    %d1,%d0                  #    Re-add size
         bne.s    exclong                  #    EQ means done.
         move.l   (%sp)+,%d2               #    Restore %d2
         rts                               #    back
exclong: move.l   (%a0),%d2                #    Fetch,
         move.l   (%a1),(%a0)+             #          and
         move.l   %d2,(%a1)+               #             exchange
         sub.l    &4,%d0                   #    Count
         bne.s    exclong                  #          down
exccomp: move.l   (%sp)+,%d2               #    Fix what we broke
         rts                               #    Return to caller
                                           #
excbyte: sub.l    &SIZEB,%d0               #    Subtract size 1st
         bmi.s    excres                   #    Copy residual
                                           #0
         move.b   (%a0),%d2                #    Fetch
         move.b   (%a1),(%a0)+          #         and
         move.b   %d2,(%a1)+            #            exchange
                                        #1
         move.b   (%a0),%d2             #    Fetch
         move.b   (%a1),(%a0)+          #         and
         move.b   %d2,(%a1)+            #            exchange
                                        #2
         move.b   (%a0),%d2             #    Fetch
         move.b   (%a1),(%a0)+          #         and
         move.b   %d2,(%a1)+            #            exchange
                                        #3
         move.b   (%a0),%d2             #    Fetch
         move.b   (%a1),(%a0)+          #         and
         move.b   %d2,(%a1)+            #            exchange
                                        #4
         move.b   (%a0),%d2             #    Fetch
         move.b   (%a1),(%a0)+          #         and
         move.b   %d2,(%a1)+            #            exchange
                                        #5
         move.b   (%a0),%d2             #    Fetch
         move.b   (%a1),(%a0)+          #         and
         move.b   %d2,(%a1)+            #            exchange
                                        #6
         move.b   (%a0),%d2             #    Fetch
         move.b   (%a1),(%a0)+          #         and
         move.b   %d2,(%a1)+            #            exchange
                                        #7
         move.b   (%a0),%d2             #    Fetch
         move.b   (%a1),(%a0)+          #         and
         move.b   %d2,(%a1)+            #            exchange
                                        #
         bra.s    excbyte               #    And around again
excres:  addq.l   &SIZEB,%d0            #    Restore
         beq.s    exccomp               #    Completed.
exc1:    move.b   (%a0),%d2             #    Fetch,
         move.b   (%a1),(%a0)+          #         and
         move.b   %d2,(%a1)+            #            exchange
         subq.l   &1,%d0                #    And
         bne.s    exc1                  #         loop
         move.l   (%sp)+,%d2            #    Fix what we broke
         rts                            #    Finished.
```

bsearch Routine. The binary search routine translates to A/UX as follows:

```
         global   bsearch
bsearch: link     %a6,&0                #   Preserve stack frames
#        movem.l  %d2-%d6/%a4,-(%sp)    #   Save registers we want
         movem.l  &0x3e08,-(%sp)        #   Save registers we want
#        movem.l  8(%a6),%d3/%d4/%d5/%d6/%a4 #   key,base,nmemb,size,compar
         movem.l  8(%a6),&0x1078        #   key,base,nmemb,size,compar
         sub.l    1,%d5                 #   Calculate
         mulu.l   %d6,%d5               #   hi = (nmemb-1)*size
         add.l    %d4,%d5               #                   + lo
                                        #   start while loop
         bra.s    bs40                  #   With conditional last
bs10:    move.l   %d5,%d2               #   Mid = hi -
         sub.l    %d4,%d2               #            lo
         asr.l    &1,%d2                #   / 2
         divs.l   %d6,%d2               #   / size
         muls.l   %d6,%d2               #   * size
         add.l    %d4,%d2               #   + lo;
#        movem.l  %d2/%d3,-(%sp)        #   Push parms
         movem.l  &0x3000,-(%sp)        #   Push parms
         jsr      (%a4)                 #   call compare func.
         addq.l   &8,%sp                #   Pop stack
         tst.l    %d0                   #   Check sign
         bgt.s    bs20                  #   mid > key
         bmi.s    bs30                  #   mid < key
         move.l   %d2,%d0               #   mid = key
         move.l   %d0,%a0               #   For picky C compilers
#        movem.l  (%sp)+,%d2-%d6/%a4    #   Pop registers
         movem.l  (%sp)+,&0x107c        #   Pop registers
         unlk     %a6                   #   restore %a6
         rts                            #   and return
                                        #
bs20:    move.l   %d2,%d5               #   hi = mid -
         sub.l    %d6,%d5               #            size
         bra.s    bs40                  #   Do loop termination code
bs30:    move.l   %d2,%d4               #   lo = mid +
         add.l    %d6,%d4               #            size
                                        #
bs40:    cmp.l    %d5,%d4               #   lo : hi
         bhs.s    bs10                  #   hi >= lo if true
         move.l   &0,%d0                #   Null pointer
         move.l   %d0,%a0               #    For picky C compilers
#        movem.l  (%sp)+,%d2-%d6/%a4    #    Pop registers
         movem.l  (%sp)+,&0x107c        #    Pop registers
         unlk     %a6                   #    restore %a6
         rts                            #    and return
```

line Routine. The line() routine (Bresenham line algorithm) translates to A/UX as follows:

```
         set      x1,8                  #    Stack offset for x1
         set      y1,12                 #    Stack offset for y1
         set      x2,16                 #    Stack offset for x2
         set      y2,20                 #    Stack offset for y2
         set      value,24              #    Stack offset for value
         global   line                  #
         global   bitmap                #    bitmap target area
line:    link     %a6,&0                #    Use standard stack frames
#        movem.l  %d2-%d7,-(%sp)        #    Save data registers.
         movem.l  &0x3F00,-(%sp)        #    Save data registers.
         move.l   bitmap,%a0            #    %a0 -> Bitmap data
         move.l   value(%a6),%a1        #    %a1 =  pixel value
         move.l   x1(%a6),%d0           #    Calculate
         sub.l    x2(%a6),%d0           #            dx
         bge.s    li10                  #    already positive
         neg.l    %d0                   #    make positive
li10:                                   #
         move.l   y1(%a6),%d1           #    Calculate
         sub.l    y2(%a6),%d1           #            dy
         bge.s    li20                  #    already positive
         neg.l    %d1                   #    make positive
li20:                                   #
         move.l   %d1,%d4               #    incr2 = dy
         sub.l    %d0,%d4               #             - dx
         move.l   %d4,%d5               #    d = dy - dx
         add.l    %d4,%d4               #    incr2       << 1
         add.l    %d1,%d5               #              + dy
         move.l   %d1,%d3               #    incr1 = dy
         add.l    %d1,%d3               #              + dy
         move.l   x2(%a6),%d6           #    x    = x2;
         cmp.l    %d6,x1(%a6)           #    if(x1 > x2)
         bge.s    li30                  #    not the case
         move.l   y2(%a6),%d7           #    y    = y2;
         move.l   x1(%a6),%d2           #    xend = x1;
         bra.s    li40                  #    branch around next
li30:    move.l   x1(%a6),%d6           #    x    = x1;
         move.l   y1(%a6),%d7           #    y    = y2;
         move.l   x2(%a6),%d2           #    xend = x1;
li40:                                   #
```

```
         move.l   %d6,%d0               #    pixel(x,
         move.l   %d7,%d1               #          y,value);
         bsr.s    pixel                 #    do it.
                                        #
         bra.s    li80                  #    use inverted loop
li50:    addq.l   &1,%d6                #    x++;
         tst.l    %d5                   #    if(d < 0)
         bge.s    li60                  #    it isn't
         add.l    %d3,%d5               #    d += incr1
         bra.s    li70                  #    skip next
li60:    addq.l   &1,%d7                #    y++;
         add.l    %d4,%d5               #    d += incr2;
li70:    move.l   %d6,%d0               #    pixel(x,
         move.l   %d7,%d1               #          y,value);
         bsr.s    pixel                 #    draw the pixel
li80:    cmp.l    %d2,%d6               #    while(x < xend)
         bge.s    li50                  #    xend >= x.
#        movem.l  (%sp)+,%d2-%d7        #    Fix what we broke
         movem.l  (%sp)+,&0x00FC        #    Fix what we broke
         unlk     %a6                   #    Pop frame
         rts                            #    Back to caller

         global   pixel                 #    Write a pixel
pixel:   move.l   %d2,-(%sp)            #    Save registers
         lsl.l    &5,%d1                #    Formulate index
         move.l   %d0,%d2               #    Copy x value
         lsr.l    &3,%d2                #    Divide by 8
         add.l    %d2,%d1               #    Byte-level index
         andi.l   &7,%d0                #    Calculate bit number
         move.b   &7,%d2                #    And
         sub.b    %d0,%d2               #        invert ordering
         cmp.l    %a1,&0                #    Set or clear?
         beq.s    clrpix                #    Clear
         bset     %d2,0(%a0,%d1.l)      #    Set the pixel
         move.l   (%sp)+,%d2            #    Restore %d2
         rts                            #    and return
clrpix:  bclr     %d2,0(%a0,%d1.l)      #    Clear the pixel
         move.l   (%sp)+,%d2            #    Restore %d2
         rts                            #    and return
```

circle Routine. The circle() routine (Bresenham circle algorithm) translates to A/UX as follows:

```
         global   circle
circle:  link     %a6,&0                #    Setup stack frame
#        movem.l  %d2-%d7,-(SP)         #    Save Registers
         movem.l  &0x3F00,-(%sp)        #    Save Registers
```

```
#        movem.l  8(%a6),d4/d5/d7/a1     #   Load arguments (y=r)
         movem.l  8(%a6),&0x02B0         #   Load arguments (y=r)
         clr.l    %d6                    #   x = 0;
         move.l   %d7,%d3                #   d = r
         add.l    %d3,%d3                #         + r
         neg.l    %d3                    #   - r+r
         addq.l   &3,%d3                 #   3 - r+r
                                         #
         bra.s    ci40                   #   top of while()
ci10:    bsr.s    circle_points          #   Do the circle stuff
         tst.l    %d3                    #   if(d < 0)
         bgt.s    ci20                   #   it isn't
         move.l   %d6,%d0                #   Temp.
         asl.l    &2,%d0                 #   x << 2
         add.l    %d0,%d3                #   d += x << 2
         addq.l   &6,%d3                 #             + 3
         bra.s    ci30                   #   skip the next part
ci20:    move.l   %d6,%d0                #   x
         sub.l    %d7,%d0                #     - y
         asl.l    &2,%d0                 #         << 2
         add.l    %d0,%d3                #   d += (x-y)<<2
         addq.l   &8,%d3                 #               +
         addq.l   &2,%d3                 #                 10
         subq.l   &1,%d7                 #   y--;
ci30:    addq.l   &1,%d6                 #   x++;
ci40:    cmp.l    %d7,%d6                #   while(x < y)
         bgt.s    ci10                   #   restart loop
         cmp.l    %d7,%d6                #   if(x == y)
         bne.s    ci50                   #   it isn't
         bsr.s    circle_points          #   do the 8-way stuff
#ci50:   movem.l  (%sp)+,%d2-%d7         #   Unsave registers
ci50:    movem.l  (%sp)+,&0x00FC         #   Unsave registers
         unlk     %a6                    #   undo stack frame
         rts                             #   and return

         global   pixel                  #   import the pixel routine
circle_points:                           #
         move.l   %d4,%d0                #1  pixel(xc
         add.l    %d6,%d0                #         +x,
         move.l   %d5,%d1                #            yc
         add.l    %d7,%d1                #              +y);
         jsr      pixel                  #   do the call
         move.l   %d4,%d0                #2  pixel(xc
         add.l    %d7,%d0                #         +y,
         move.l   %d5,%d1                #            yc
         add.l    %d6,%d1                #              +x);
         jsr      pixel                  #   do the call
```

```
        move.l   %d4,%d0                   #3   pixel(xc
        add.l    %d7,%d0                   #      +y,
        move.l   %d5,%d1                   #         yc
        sub.l    %d6,%d1                   #           -x);
        jsr      pixel                     #    do the call
        move.l   %d4,%d0                   #4   pixel(xc
        add.l    %d6,%d0                   #          +x,
        move.l   %d5,%d1                   #             yc
        sub.l    %d7,%d1                   #               -y);
        jsr      pixel                     #    do the call
        move.l   %d4,%d0                   #5   pixel(xc
        sub.l    %d6,%d0                   #          -x,
        move.l   %d5,%d1                   #             yc
        sub.l    %d7,%d1                   #               -y);
        jsr      pixel                     #    do the call
        move.l   %d4,%d0                   #6   pixel(xc
        sub.l    %d7,%d0                   #          -y,
        move.l   %d5,%d1                   #             yc
        sub.l    %d6,%d1                   #               -x);
        jsr      pixel                     #    do the call
        move.l   %d4,%d0                   #7   pixel(xc
        sub.l    %d7,%d0                   #          -y,
        move.l   %d5,%d1                   #             yc
        add.l    %d6,%d1                   #               +x);
        jsr      pixel                     #    do the call
        move.l   %d4,%d0                   #8   pixel(xc
        sub.l    %d6,%d0                   #          -x,
        move.l   %d5,%d1                   #             yc
        add.l    %d7,%d1                   #               +y);
        jsr      pixel                     #    do the call
        rts                                #    return
```

bisect Routine. The bisect() routine translates to A/UX as follows. Note that you cannot use the FMOVEM instruction to load the parameters in registers, as A/UX passes floating point numbers in double, rather than extended, precision.

```
        global   bisect
bisect: link     %a6,&0                    #    Preserve stack frames
        move.l   %a4,-(%sp)                #    Save integer registers
#       fmovem   %fp2-%fp7,-(%sp)          #    Save Floating registers
        fmovem   &0x00fc,-(%sp)            #    Save Floating registers
        move.l   8(%a6),%a4                #    Function pointer
        fmove.d  12(%a6),%fp2              #    Load
        fmove.d  20(%a6),%fp3              #         up
        fmove.d  28(%a6),%fp4              #            arguments
        fmove.d  %fp2,-(%sp)               #    Calculate f(a)
        jsr       (%a4)                 #   call f
        fmove.x   %fp0,%fp5             #   Save in fa
        add.l     &8,%sp                #   Pop argument
        fmove.d   %fp3,-(%sp)           #   Now get
        jsr       (%a4)                 #          f(b)
        fmove.x   %fp0,%fp6             #   Save in fb
        add.l     &8,%sp                #   Pop argument
        fmul.x    %fp5,%fp0             #   fa * fb...
        fbnge     bi10                  #   branch if negative
#       fmovem    (%sp)+,%fp2-%fp7      #   Pop floating regs
        fmovem    (%sp)+,&0x003f        #   Pop floating regs
        move.l    (%sp)+,%a4            #   Pop integer  regs
        fmove.b   &0,%fp0               #   Return zero
        unlk      %a6                   #   Restore A6
        rts                             #
                                        #
bi10:   bra.s     bi40                  #   While loop begins
bi20:   fmove.x   %fp2,%fp7             #   mid =
        fadd.x    %fp3,%fp7             #         a + b
        fdiv.b    &2,%fp7               #               /2
        fmove.d   %fp7,-(%sp)           #   Push argument
        jsr       (%a4)                 #   call f
        fmove.x   %fp0,%fp1             #   Save result (temp)
        fmul.x    %fp5,%fp1             #   f(a) * f(mid)
        add.l     &8,%sp                #   Pop argument to f()
        fbnle     bi30                  #   it's positive
        fmove.x   %fp7,%fp3             #   b  = mid
        fmove.x   %fp0,%fp6             #   fb = fmid
        bra.s     bi40                  #   Now test termination
bi30:   fmove.x   %fp7,%fp2             #   a  = mid
        fmove.x   %fp0,%fp5             #   fa = fmid
                                        #
bi40:   fmove.x   %fp2,%fp1             #   Calculate
        fsub.x    %fp3,%fp1             #           a - b
        fabs.x    %fp1                  #   take absolute value
        fcmp.x    %fp1,%fp4             #   eps : abs(a-b)
        fbgt      bi20                  #   if (a-b) > eps, loop
        fmove.x   %fp7,%fp0             #   Set return value
#       fmovem    (%sp)+,%fp2-%fp7      #   Pop floating registers
        fmovem    (%sp)+,&0x003f        #   Pop floating registers
        move.l    (%sp)+,%a4            #   Pop procedure pointer
        unlk      %a6                   #   unframe
        rts                             #   return to caller
```

gauss() Routine. The Gaussian elimination function translates as follows:

```
         global   gauss
         set      NMAX,50
gauss:   link     %a6,&0                 #    Preserve stack frames
#        movem.l  %d2-%d7/%a2-%a4,-(%sp) #    Save integer registers
         movem.l  &0x3f38,-(%sp)         #    Save integer registers
#        fmovem.x %fp2/%fp3,-(%sp)       #    Save Floating registers
         fmovem.x &0x000c,-(%sp)         #    Save Floating registers
#        movem.l  8(%a6),%a0/%a1/%a2     #    Arguments
         movem.l  8(%a6),&0x0700         #    Arguments
         move.l   %a2,%d4                #    (n belongs in data register)
         move.l   &NMAX+1,%d7            #    For indexing
                                         #
         clr.l    %d2                    #    k = 0
ga10:    move.l   %d4,%d5                #    Calculate n
         subq.l   &1,%d5                 #               - 1
         cmp.l    %d5,%d2                #    k : n-1
         ble      ga100                  #    loop exceeded
                                         #
         move.l   %d2,%d3                #    l = k;
                                         #
         move.l   %d2,%d0                #    for(i=k
         addq.l   &1,%d0                 #          +1;
         move.l   %d0,%d5                #    Compute
         mulu.l   %d7,%d5                #          a[i][k]
         add.l    %d2,%d5                #                 index
         move.l   %d3,%d6                #    Compute
         mulu.l   %d7,%d6                #          a[l][k]
         add.l    %d2,%d6                #                 index
                                         #
ga20:    cmp.l    %d4,%d0                #              i < n;
         ble      ga40                   #    (i >= n) -- terminate
                                         #
         fmove.d  0(%a0,%d5.l*8),%fp0    #    a[i][k]
         fabs.x   %fp0                   #    take fabs(a[i][k])
                                         #
         fmove.d  0(%a0,%d6.l*8),%fp1    #    a[l][k]
         fabs.x   %fp1                   #    take fabs(a[i][k])
                                         #
         fcmp.x   %fp1,%fp0              #    a[i][k] : a[l][k]
         fbnle    ga30                   #    don't move l
         move.l   %d0,%d3                #    l = i
         move.l   %d5,%d6                #    Copy index as well
ga30:    add.l    %d7,%d5                #    Bump up index
         addq.l   &1,%d0                 #                  i++)
         bra      ga20                   #    Back to loop top
ga40:                                    #    here when loop terminates
         move.l   %d2,%d5                #    Compute
         mulu.l   %d7,%d5                #          a[k]
         add.l    %d2,%d5                #                [k]
         lea      0(%a0,%d5.l*8),%a2     #    %a2 -> a[k][k]
         move.l   %a2,%a4                #    Save for later
         cmp.l    %d2,%d3                #    if(l != k)
         beq      ga55                   #    Equal, don't switch
         move.l   %d3,%d5                #    Compute
         mulu.l   %d7,%d5                #    a[l]
         add.l    %d2,%d5                #                [k]
         lea      0(%a0,%d5.l*8),%a3     #    %a3 -> a[l][k]
         move.l   %d4,%d5                #    %d5 = n
         sub.l    %d2,%d5                #          - k (DBRA Count)
ga50:    move.l   (%a2),%d6              #    Exchange
         move.l   (%a3),(%a2)+           #            a
         move.l   %d6,(%a3)+             #               long
         move.l   (%a2),%d6              #    Exchange
         move.l   (%a3),(%a2)+           #            a
         move.l   %d6,(%a3)+             #               long
         dbra     %d5,ga50               #    Repeat as needed
ga55:                                    #
         fmove.d  (%a4),%fp3             #    Fetch a[k][k] once
         move.l   %d2,%d0                #    for(i = k
         addq.l   &1,%d0                 #           +1;
         move.l   %d0,%d5                #    Compute
         mulu.l   %d7,%d5                #          a[i][0]
         lea      0(%a0,%d5.l*8),%a2     #                 index
         move.l   %d2,%d6                #    Compute
         mulu.l   %d7,%d6                #          a[k][0]
         lea      0(%a0,%d6.l*8),%a3     #                 index
ga60:    cmp.l    %d4,%d0                #                i < n;
         ble      ga90                   #
         fmove.d  0(%a2,%d2.l*8),%fp1    #    %fp1 = a[i][k]
         move.l   %d2,%d1                #    j = k
         addq.l   &1,%d1                 #           + 1
         fdiv.x   %fp3,%fp1              #    %fp1 = a[i][k] / a[k][k]
ga70:                                    #    for(j=k+1;j <=n; j++)
         cmp.l    %d4,%d1                #    j : n
         blt      ga80                   #    if LT, inner loop over
         fmove.d  0(%a3,%d1.l*8),%fp0    #    %fp0 = a[k][j]
         fmul.x   %fp1,%fp0              #                * temp
         fmove.d  0(%a2,%d1.l*8),%fp2    #    %fp2 = a[i][j]
         fsub.x   %fp0,%fp2              #                - a[k][j]*temp
         fmove.d  %fp2,0(%a2,%d1.l*8)    #    Store in a[i][j]
         addq.l   &1,%d1                 #    j++
         bra      ga70                   #    Loop again
ga80:                                    #
         addq.l   &1,%d0                 #                   i++)
         lea      0(%a2,%d7.l*8),%a2     #    Bump induction pointer also
         bra      ga60                   #    Repeat the for()
ga90:                                    #    end for(i=k+1;i<n;i++)
```

```
                                           #
        addq.l   &1,%d2                    #    k++ (in for loop)
        bra      ga10                      #    repeat for loop
ga100:                                     #    end triangularization
                                           #
        move.l   %d4,%d5                   #    Calculate
        subq.l   &1,%d5                    #            n - 1
        move.l   %d5,%d6                   #    Compute
        mulu.l   %d7,%d6                   #          a[n-1][n]
        add.l    %d4,%d6                   #                 index
        fmove.d  0(%a0,%d6.l*8),%fp0       #    %fp0 = a[n-1][n]
        subq.l   &1,%d6                    #    Now a[n-1][n-1]
        fdiv.d   0(%a0,%d6.l*8),%fp0       #    %fp0 = a[n-1][n]/a[n-1][n-1]
        fmove.d  %fp0,0(%a1,%d5.l*8)       #    x[n-1] =     "
                                           #
        move.l   %d4,%d0                   #    i =
        subq.l   &2,%d0                    #        n-2;
ga110:                                     #    top of for()
        tst.l    %d0                       #    Still >= 0?
        bmi      ga140                     #    No, we're done.
        fmove.b  &0,%fp0                   #    temp = 0;
        move.l   %d0,%d1                   #    for(j=i
        addq.l   &1,%d1                    #          +1;
        move.l   %d0,%d6                   #    Calculate a[i][0]
        mulu.l   %d7,%d6                   #    For inside loop
ga120:  cmp.l    %d4,%d1                   #    j < n;
        ble      ga130                     #
        fmove.d  0(%a1,%d1.l*8),%fp1       #    %fp1 = x[j]
        move.l   %d6,%d5                   #    Calculate a[i][j]
        add.l    %d1,%d5                   #            index
        fmul.d   0(%a0,%d5.l*8),%fp1       #    x[j]*a[i][j]
        fadd.x   %fp1,%fp0                 #    temp += x[j]*a[i][j]
        addq.l   &1,%d1                    #    j++
        bra      ga120                     #    To top of for loop
ga130:  move.l   %d6,%d5                   #    Compute a[i][n]
        add.l    %d4,%d5                   #                index
        fmove.d  0(%a0,%d5.l*8),%fp2       #    %fp2 = a[i][n]
        fsub.x   %fp0,%fp2                 #                   -temp
        move.l   %d6,%d5                   #    Compute a[i][i]
        add.l    %d0,%d5                   #                index
        fdiv.d   0(%a0,%d5.l*8),%fp2       #    %fp2 = (a[i][n]-temp)/a[i][i]
        fmove.d  %fp2,0(%a1,%d0.l*8)       #    put in x[i]
        subq.l   &1,%d0                    #    Decrement i
        bra      ga110                     #    Back to loop top
ga140:                                     #    end of procedure
#       fmovem.x (%sp)+,%fp2/%fp3          #    Restore floating registers
        fmovem.x (%sp)+,&0x0030            #    Restore floating registers
#       movem.l  (%sp)+,%d2-%d7/%a2-%a4 #    Restore integer registers
```

```
         movem.l  (%sp)+,&0x1cfc         #    Restore integer registers
         unlk     %a6                    #    unframe
         rts                             #    Return to caller
```

is020 Routine. The A/UX version of the is020() routine is as follows:

```
         global   is020
is020:   move.l   &1,%d0                 #    Set index register
         lea      table,%a0              #->  table
         move.b   0(%a0,%d0.l*4),%d0     #    get return value
         rts                             #    Back to caller
table:   byte     0,0,0,0                #    Return value table
         byte     1,0,0,0                #
```

config Routine. The config() routine cannot actually run under A/UX due to the need to execute privileged instructions and access the vector table. The translation is presented here for completeness. You could use this code if you are using UNIX as a development host for some target machine where your software runs in supervisor mode. UNIX is often used as the development platform for embedded controllers, for instance. The config() routine written for the A/UX assembler is as follows:

```
         set      IS000,0                #         0 => 68000
         set      IS010,1                #         1 => 68010
         set      IS020,2                #         2 => 68020
         set      IS030,3                #         3 => 68030
                                         #
         set      IS881,0x100            #       100 => 68881 present
         set      IS882,0x200            #       200 => 68882 present
                                         #
         set      MMU,0x10000            #    100000 => MMU is present
                                         #
         set      illvec,0x010           #    Illegal instruction vector
         set      linef,0x2C             #    Line F vector

         global   config
config:  move.l   &1,%d0                 #    Set index register
         lea      table,%a0              #    %a0 -> table
         move.b   0(%a0,%d0.l*4),%d0     #    get return value
         bne.s    got020                 #    It's an 020.
         lea      illvec,%a0             #    %a0-> Illegal inst vector
         lea      con20,%a1              #    %a1-> Trap handler
         move.l   (%a0),-(%sp)           #    Save old vector
         move.l   %a1,(%a0)              #    Set new vector
         move.l   %sp,%a1                #    Save stack pointer
         move.w   %cc,%d0                 #    See if 010.
con10:   move.l   &IS010,%d0              #    It's an 010.
con20:   move.l   %a1,%sp                 #    Restore stack
         move.l   (%sp)+,(%a0)            #    Restore vector
         rts                              #    %d0 = IS000 or IS010
                                          #
got020:  move.l   %vbr,%a0                #    Get vector base register
         lea      linef(%a0),%a0          #    %a0-> Line F trap vector
         move.l   (%a0),-(%sp)            #    Save old vector
         lea      con40,%a1               #    %a1-> Trap handler
         move.l   %a1,(%a0)               #    Set the vector
         move.l   %sp,%a1                 #    Remember stack pointer
         fnop                             #    Make sure it's idle
         fsave    -(%sp)                  #    See if got FPU
         cmp.b    1(%sp),&0x18            #    Is it a 68881?
         beq.s    con30                   #    No.
         add.l    &IS881,%d0              #    Make a 68882
con30:   add.l    &IS881,%d0              #    or a 68881
con40:   move.l   %a1,%sp                 #    clean up stack
         lea      con20,%a1               #    Set vector again
         move.l   %a1,(%a0)               #    point to exit routine
         move.l   %sp,%a1                 #    Save stack pointer
#        pflusha                          #    Flush MMU
         short    0xF000,0x2400           #    They really should fix this...
         add.l    &MMU,%d0                #    Set MMU flag
         lea      con50,%a1               #    Set another vector
         move.l   %a1,(%a0)               #    Now see if 030
         move.l   %sp,%a1                 #    Remember stack
#        psave    -(%sp)                  #    Only for 851
         short    0xF127                  #    They really should fix this...
         bra.s    con20                   #    Go exit
con50:   or.l     &IS030,%d0              #    We're an 030
         bra.s    con20                   #    Go exit
table:   byte     0,0,0,0                 #    CPU value table
         byte     2,0,0,0                 #
```

There are two things to note about this code. First, the Condition Code Register (CCR) is %cc when writing for A/UX. Second, the PMMU instructions do not assemble (the manual claims otherwise, but don't believe it). You must hand-assemble the instructions and put the hex values in your code with the short directive, as shown above. A useful technique here is to get an assembly listing of equivalent code on another system. We used the MPW assembler to check this hand assembly and also to generate the register masks for MOVEM and FMOVEM instructions.

tracer Routine. The tracer() routine also cannot run under A/UX since it modifies the vector table and uses privileged instructions.

The 68010/020/030 version is as follows:

```
         set      XD0,0                  #   %d0 offset
         set      XA0,8*4                #   %a0 offset
         set      XSP,15*4               #   Stack pointer offset
         set      XPC,XSP+4              #   Program counter offset
         set      XSR,XPC+4              #   Status register offset
         set      NTRAPS,256             #   Number of vectors
         set      TRACE,0x8000           #   Trace bit
         set      RTEINST,0x4E73         #   RTE instruction word
         global   tracer
tracer:  move.l   4(%sp),%a0             #   %a0 -> Register area
         lea      dest,%a1               #   %a1 -> Destination reg area
         move.l   8(%sp),(%a1)           #   Remember this address
#        movem.l  %d2-%d7/%a2-%a6,-(%sp) #   Save registers
         movem.l  &0x3F3E,-(%sp)         #   Save registers
         lea      oldsp,%a1              #   %a1 -> %sp Save area
         move.l   %sp,(%a1)              #   Save this
         move.l   XSP(%a0),%sp           #   Switch to his stack
         clr.w    -(%sp)                 #010Format word
         move.l   XPC(%a0),-(%sp)        #   Set PC
         move.l   (%sp),%a1              #   %a1 -> Target instruction
         move.w   XSR+2(%a0),-(%sp)      #   And SR
         ori.w    &TRACE,(%sp)           #   Set trace bit
         cmp.w    (%a1),&RTEINST         #   An RTE Instruction?
         bne.s    NOTRTE                 #   No.
         ori.w    &TRACE,8(%sp)          #010Set trace in SR for RTE
NOTRTE:  bsr.s    trapsin                #   Install exception hooks
#        movem.l  (%a0),%d0-%d7/%a0-%a6  #   Set registers
         movem.l  (%a0),&0x7FFF          #   Set registers
         rte                             #   And trace it.
oldsp:   long     0                      #   Old stack pointer
dest:    long     0                      #-> Destination area
         global   except                 #   Exception handler (010-030)
except:  move.l   %a0,-(%sp)             #   Save the
         move.l   dest,%a0               #   %a0 -> Destination area
#        movem.l  %d0-%d7/%a0-%a6,(%a0)  #   Save the registers
         movem.l  &0x7FFF,(%a0)          #   Save the registers
         move.l   (%sp)+,XA0(%a0)        #   Save %a0
         move.w   (%sp)+,XSR+2(%a0)      #   save status register
         move.l   (%sp)+,XPC(%a0)        #   And program counter
         andi.w   &TRACE-1,XSR+2(%a0)    #   Clean the trace bit
         move.w   (%sp)+,%d0             #010Get format word
         move.w   %d0,%d1                #010Isolate
         lsr.w    &8,%d1                 #010      Format bits
         lsr.w    &4,%d1                 #010      Format bits
         move.b   bytes(%d1.W),%d1       #010Fetch frame size
         add.l    %d1,%sp                #   Trim stack
         move.l   %sp,XSP(%a0)           #   Save stack
         move.l   oldsp,%sp              #    Fetch old stack pointer
         bsr.s    trapsout               #    Take traps out
#        movem.l  (sp)+,d2-d7/a2-a6      #    Restore registers
         movem.l  (%sp)+,&0x7CFC         #    Restore registers
         andi.l   &0xFFF,%d0             #    Strip to 12 bits
         lsr.w    &2,%d0                 #    Convert to vector number
         rts                             #    Done!
bytes:   byte     00,00,04,00,00,00,00,00 # 0-7  Number of bytes
         byte     50,12,24,84,00,00,00,00 # 8-F              to pop
         global   trapsin                #    Put traps in place
trapsin: move.l   %vbr,%a2               #    %a2 -> Vector area
         lea      except,%a3             #    %a3 -> New exception handler
         move.w   &NTRAPS-1,%d1          #    %d1 = & traps to do
         lea      traploc,%a1            #    %a1 -> Save area
inloop:  move.l   (%a2),(%a1)+           #    Save trap
         move.l   %a3,(%a2)+             #    Set new trap
         dbra     %d1,inloop             #    Count down
         rts                             #    Go back to beginning
         global   trapsout               #    Take traps back out
trapsout: move.l  %vbr,%a2               #    %a2 -> Vectors
         lea      traploc,%a1            #    %a1 -> Save area
         move.w   &NTRAPS-1,%d1          #    %d1 = & traps to do
outloop: move.l   (%a1)+,(%a2)+          #    Restore
         dbra     %d1,outloop            #    and loop
         rts                             #    Done.
         global   traploc                #    Trap location
traploc:                                 #
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 00-0F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 10-1F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 20-2F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 30-3F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 40-4F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 50-5F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 60-6F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 70-7F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 80-8F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 90-9F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # A0-AF
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # B0-BF
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # C0-CF
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # D0-DF
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # E0-EF
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # F0-FF
```

The 68000 version of this routine is as follows:

```
         set      XD0,0                  #    D0 offset
         set      XA0,8*4                #    A0 offset
         set      XSP,15*4                #     Stack pointer offset
         set      XPC,XSP+4               #     Program counter offset
         set      XSR,XPC+4               #     Status register offset
         set      NTRAPS,256              #     Number of vectors
         set      TRACE,0x8000            #     Trace bit
         set      RTEINST,0x4E73          #     RTE instruction word
         global   tracer
tracer:  move.l   4(%sp),%a0              #     %a0 -> Register area
         lea      dest,%a1                #     %a1 -> Destination reg area
         move.l   8(%sp),(%a1)            #     Remember this address
#        movem.l  %d2-D7/%a2-%a6,-(%sp)   #     Save registers
         movem.l  &0x3F3E,-(%sp)          #     Save registers
         lea      oldsp,%a1               #     %a1 -> %sp Save area
         move.l   %sp,(%a1)               #     Save this
         move.l   XSP(%a0),%sp            #     Switch to his stack
         move.l   XPC(%a0),-(%sp)         #     Set PC
         move.l   (%sp),%a1               #     %a1 -> Target instruction
         move.w   XSR+2(%a0),-(%sp)       #     Set SR
         ori.w    &TRACE,(%sp)            #     Set trace bit
         cmp.w    (%a1),&RTEINST          #     An RTE Instruction?
         bne.s    NOTRTE                  #     No.
         ori.w    &TRACE,6(%sp)           #000  Set trace in SR for RTE
NOTRTE:  bsr.s    trapsin                 #     Install exception hooks
#        movem.l  (%a0),%d0-D7/%a0-%a6    #     Set registers
         movem.l  (%a0),&0x7FFF           #     Set registers
         rte                              #     And trace it.
oldsp:   long     0                       #     Old stack pointer
dest:    long     0                       #->   Destination area
         global   except                  #     Exception handler (010-030)
except:  move.l   %a0,-(%sp)              #     Save the
         move.l   dest,%a0                #     %a0 -> Destination area
#        movem.l  %d0-D7/%a0-%a6,(%a0)    #     Save the registers
         movem.l  &0x7FFF,(%a0)           #     Save the registers
         move.l   (%sp)+,XA0(%a0)         #     Save %a0
         bsr.s    getpc                   #000  fetch PC
         nop                              #000  Quieten the assembler
getpc:   move.l   (%sp)+,%d0              #000  %d0 = Our PC
         clr.w    %d0                     #000  Clear low word
         swap     %d0                     #000  Isolate
         lsr.w    &0x8,%d0                #000             top byte
         cmp.b    %d0,&2                  #000  See if BUSERR
         blt.s    noadj                   #000  Don't adjust %sp
         cmp.b    %d0,&3                  #000  See if Address error
         bgt.s    noadj                   #000  Don't adjust
         add.l    &8,%sp                  #000  Pop stack stuff
noadj:   move.w   (%sp)+,XSR+2(%a0)       #     save status register
         move.l   (%sp)+,XPC(%a0)         #     And program counter
         andi.w   &TRACE-1,XSR+2(%a0)     #     Clean the trace bit
         move.l   %sp,XSP(%a0)           #     Save stack
         move.l   oldsp,%sp              #     Fetch old stack pointer
         bsr.s    trapsout               #     Take traps out
#        movem.l  (%sp)+,%d2-D7/%a2-%a6  #     Restore registers
         movem.l  (%sp)+,&0x7CFC         #     Restore registers
         rts                             #     done!

         global   trapsin                #     Put traps in place
trapsin: sub.l    %a2,%a2                #     %a2 -> Vector area
         lea      except,%a3             #     %a3 -> New exception handler
         move.w   &NTRAPS-1,%d1          #     %d1 = # traps to do
         clr.l    %d2                    #     %d2 = counter
         lea      traploc,%a1            #     %a1 -> Save area
inloop:  move.l   (%a2),(%a1)+           #     Save trap
         move.l   %a3,(%a2)              #     Set new trap
         move.b   %d2,(%a2)              #     Set trap ID
         addq.l   &1,%d2                 #     Fix %d2
         addq.l   &4,%a2                 #     Advance %a2
         dbra     %d1,inloop             #     Count down
         rts                             #     Go back to beginning
         global   trapsout               #     Take traps back out
trapsout: sub.l   %a2,%a2                #     %a2 -> Vectors
         lea      traploc,%a1            #     %a1 -> Save area
         move.w   &NTRAPS-1,%d1          #     %d1 = # traps to do
outloop: move.l   (%a1)+,(%a2)+          #     Restore
         dbra     %d1,outloop            #     and loop
         rts                             #     Done.
         global   traploc                #     Trap location
traploc: #
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 00-0F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 10-1F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 20-2F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 30-3F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 40-4F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 50-5F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 60-6F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 70-7F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 80-8F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # 90-9F
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # A0-AF
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # B0-BF
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # C0-CF
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # D0-DF
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # E0-EF
         long     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 # F0-FF
```

Appendix B
Glossary

"A powerful agent is the right word."
— Mark Twain

Introduction

This appendix contains definitions of technical terms used in this book and in the computer industry in general.

Definitions

One problem with technical terms is that definitions depend on environment. Thus, terms sometimes mean different things to different people. Our view is represented in these definitions.

80x86. A line of CPU chips made by Intel. These chips (8088, 8086, 80188, 80186, 80286, and 80386) are used in the IBM PC and compatible forms of personal computers.

68000. A Central Processing Unit (CPU) chip with a 16-bit data bus and 24-bit address bus. The 68000 is the original member of the Motorola 680x0 family. You can find this chip in the Atari ST, Amiga 500/2000 series, Macintosh 128/512/Plus/SE, and in several other computers.

68008. A 68000 chip with an 8-bit data bus.

68010. The successor to the 68000. This chip also has a 16-bit data bus and 24-bit address bus. This is the first 680x0 chip to support virtual memory (using instruction continuation). This chip is in the Sun II and several other UNIX-based products.

68020. The successor to the 68010. This chip has a 32-bit data bus and a 32-bit address bus. This chip is the first to support hardware floating point (see 68881) and a demand paged MMU (see 68851). This chip is in the 68020 in the Macintosh II, Sun III, the Apple LaserWriter II/NTX, and other products.

68030. The successor to the 68020. This chip has a subset of the 68020 MMU on board, making it more cost effective than the 68020 for high-end workstation products. This chip is in the Macintosh IIx, Macintosh SE/30, the NeXT machine, and countless other workstation products.

68451. An MMU intended for use with the 68010.

68851. The 68851 is the Memory Management Unit (MMU) coprocessor for the 68020. This chip implements virtual memory, fine-grained access protection, and hardware breakpoint features.

68881. The 68881 is the Floating Point coprocessor for the 68020 and 68030. This chip implements IEEE floating point, as described in Chapter 4.

68882. The 68882 is the successor to the 68881. This chip is instruction-set compatible with but faster than the 68881. It is described in Chapter 4.

A/UX. A derivative of the AT&T UNIX operating system, which runs on Apple Macintosh II hardware.

Absolute Long. A 68000 addressing mode where the operand is in memory. A 32-bit longword following the instruction word contains the address of the operand. Although this mode is widely used on most 680x0 family computers as the primary way to address data memory, it is not supported by the Macintosh operating system.

Absolute Short. A 68000 addressing mode where the operand is in memory. A 16-bit word following the instruction word is sign-extended to 32 bits, and is then used as the address.

AC. See Access Control Register.

Access Control. A 68851 feature that creates eight privilege levels so that memory protection can be very tightly regulated. This feature was abandoned on the 68030.

Access Control Register. A 68851 16-bit register that regulates the access control feature.

Address Error. A fault that results from attempting to access memory on an invalid boundary. Address Error Exceptions use vector 3, with a 14-byte stack frame on the 68000, format 8 on the 68010, and formats 10 ($A) and 11 ($B) on the 68020 and later chips.

Address Register. One of eight 32-bit registers in 68000 family chips. You can use word and long sections of the register, but all 32 bits are always affected (using sign extension). Address registers are named A0–A7. Register A7 (also called SP) is used as the stack pointer for interrupts and subroutine calls.

Address Register Direct. A 68000 addressing mode where the operand is in an address register.

Address Register Indirect. A 68000 addressing mode where the operand is in a memory location whose address is in an address register.

Address Register Indirect with Displacement. A 68000 addressing mode where the operand is in a memory location whose address is specified by an address register plus a 16-bit sign extended displacement.

Address Register Indirect with Index (8-bit Displacement). A 68000 addressing mode where the operand is in a memory location whose address is specified by an address register plus an index register plus a sign-extended 8-bit displacement. The index register may be either an address or data register. You can use either the low-order 16 bits or all 32 bits of the register. On 32-bit CPUs, the contents of the register can be multiplied by 1, 2, 4, or 8. This scaling feature is not implemented on the 68000 or 68010.

Address Register Indirect with Index (Base Displacement). A 68020 addressing mode where the operand is in a memory location. The address of the operand is calculated as the contents of an address register plus the contents of a scaled index register plus a 16- or 32-bit displacement (16-bit displacements are sign-extended to 32 bits before use). The index register contents can be scaled by 1, 2, 4, or 8 before use (the index register itself is not modified). This mode is not implemented on the 68000 or 68010.

Address Register Indirect with Postincrement. A 68000 addressing mode where the operand is in a memory location whose address is in an address register. The register is incremented by the size of the operand after use.

Address Register Indirect with Predecrement. A 68000 addressing mode where the operand is in a memory location whose address is in an address register. The register is decremented by the size of the operand before use.

Address Translation Cache. (ATC) The heart of a modern MMU, the ATC is also called a Translation Lookaside Buffer. Memory is split into very small segments, with the translation table also in memory. The ATC contains the most recently used translations. This dramatically reduces the need to access memory for translation information.

Addressing Modes. The 68000 family features several different ways of specifying the location of the data to be used by an instruction. There is a 6-bit field (possibly with one or more 16-bit extension words) that specifies where the data is to be found. The techniques represented by this field are called addressing modes and are described in Chapter 2. (See Effective Address.)

Algorithm. A step-by-step procedure for performing some action. Programs are commonly expressed in algorithmic form before translation to a programming language. A common example is the description of graphics algorithms in most text books or of sorting routines in the Knuth volume *Fundamental Algorithms, Volume III: Sorting and Searching* (Addison-Wesley, 1973).

AND. An operation between two binary quantities, in which the result has a one bit only in positions where the two source operands both have a one bit.

Apple. One of the original personal computer vendors. The Apple Macintosh computer line uses the 68000, 68020, and 68030 chips described in this book.

Arithmetic Shift. A binary operation in which an operand is moved to the right or to the left. The sign bit is propagated if the shift is toward the right. Zeros are added on the right for a left shift.

Array. A group of identical sized data items placed one after the other in memory. Access is through an index quantity. To access a particular item, take the index, multiply by the item size, and add the address of the first item in the array.

ASCII. (American Standard Code for Information Interchange). The most widely used method for assigning byte values to alphanumeric characters. ASCII assigns hex values to characters as follows:

Lower

		0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
	0x0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
U	0x1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
p	0x2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
p	0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
e	0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
r	0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
	0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Note that the standard uses only the low 7 bits of an 8-bit byte to represent a character.

Assembler. A programming tool that takes human-readable source code in assembly language and translates it to machine code that a computer can directly execute. Oftentimes the output of the assembler must pass through a program called the linker, which can combine multiple modules into a single executable image. The MPW assembler is called ASM, and the UNIX assembler is called *as*.

Assembly Language. A programming technique that allows the programmer to write software in terms of the actual primitives (called instructions) that the computer will execute. Assembly Language must be translated by a program called an assembler, which converts the human-readable program into something the machine can directly execute. This technique allows exact control over the machine, but demands extreme attention to detail.

ATC. See Address Translation Cache.

Autovectored Interrupts. Interrupts whose vector is based on the interrupt priority. There are seven such interrupts, using vectors 25–31.

Back Substitution. The last part of a Gaussian elimination algorithm for solving simultaneous linear equations. Given a matrix in triangular form, the last equation has one unknown and is trivially solved. The next to last equation has two unknowns. By substituting the results of the previous equation, you can now solve that one. The process can be repeated until all unknowns have been resolved.

Backspace Character. An 8-bit quantity with the value 8. This character causes an output device to back up one character position horizontally.

BACR. See Breakpoint Registers.

BADR. See Breakpoint Registers.

Base Displacement. A field in 68020 addressing modes that can be either 16 or 32 bits. This field is the first (and possibly the only) displacement field in the instruction.

Base/Bounds. A primitive form of memory management, that provides protection and relocation, but very little else.

BASIC. Beginners All-purpose Symbolic Instruction Code. An early high-level language intended for educational use.

Big Endian. This term describes the ordering of bytes in memory. Big endian machines have the most-significant byte at the lowest address. The 68000, Z8000, and IBM 360/370/43xx/30xx/... CPUs use big endian byte ordering.

Binary. A numbering scheme using base 2. Each digit can have one of two values, 0 or 1 (OFF or ON). This scheme is extremely useful for building computer hardware.

Binary Coded Decimal. (BCD). A technique that uses 4 bits to represent a decimal digit. Six of the 16 possible values of the 4-bit field are ignored. This technique has been used by accounting programs for many years to eliminate the need to convert between decimal and binary.

Binary Search. A search technique that looks at the midpoint of a table of sorted items. One of the endpoints is moved and the process is repeated until the desired item is found or the search terminates.

Bisection. A technique that solves a non-linear equation by taking the midpoint of a region known to contain the root. Since one of the two areas then contains the root, you move an endpoint and repeat the process. This technique allows finding the root to any desired precision

Bit. A single BInary digiT. A bit can have one of two values: 0 (OFF) or 1 (ON).

Bit Field. A collection of 1–32 bits on an arbitrary bit boundary in memory. This concept is useful for implementing computer graphics and other bit-related applications.

Bitmap. A generic term for the image of a graphics buffer (usually a screen) in memory.

Boolean Operations. Logical operations on binary data. Each bit of such an operation can be determined by at most two bits from two sources. Examples are AND, OR, and Exclusive OR.

Branch or Set on UNordered Exception. A floating point exception that results from executing a conditional instruction with the NAN bit set in the floating point status register. This exception uses vector 48 and a format 0 or format 9 stack frame.

Breakpoint Registers. The 68851 has eight pairs of hardware breakpoint registers. These are called Breakpoint Acknowledge Control Registers (BACR0-BADR7) and Breakpoint Acknowledge Control Registers (BADR0-BADR7).

Bresenham, J. E. A computer scientist whose line and circle drawing algorithms are widely used in graphics textbooks.

BSUN. See Branch or Set on Unordered Exception.

Bug. An error in a program.

Burst Mode. A feature of the 68030 memory caches (both code and data) that fetches four longwords when filling a cache entry.

BUSERR. A fault that results from attempting to access non-existent memory. BUSERR exceptions use vector 2 with a 14-byte stack frame on the 68000, format 8 on the 68010, and formats $A and $B on the 68020 and later chips.

Byte. A collection of 8 bits. A byte can have 256 unique values.

C. A computer language originally designed for the UNIX system by Dennis Ritchie of Bell Laboratories. C is the successor to B.

C++. The successor to C. This language was also developed at Bell Laboratories by Bjarne Stroustroup and others.

CAAR. See Cache Address Register.

Cache. A technique that places a small fast memory between a CPU and a slower memory. Since many programs reference either the same data or data close together, the cache speeds up overall memory access times. This same technique is also often used for disk and network storage devices and attempts to keep in primary memory data that will be needed soon.

Cache Address Register. A 32-bit register that specifies an entry in the 68020 or 68030 memory cache.

Cache Control Register. A 32-bit register that controls operation of the 68020 and 68030 memory cache features.

CACR. See Cache Control Register.

CAL. See Current Access Level Register.

Carriage Return Character. A character with the value 13 (decimal) that causes an output device to return the cursor to the beginning of the line. The MPW C language uses this character in place of a newline character.

Carry Bit. Bit 0 of the Condition Code Register. This bit is set when an arithmetic operation has a carry or a borrow, or by the bit shifted out of an operand on a shift operation.

CCR. See Condition Code Register.

CD. (Compact Disk). An optically scanned storage device that can store binary data. The contents of this device cannot be modified after manufacturing.

Central Processing Unit. A piece of hardware that forms the heart of a computer. The CPU fetches instructions from memory and executes them.

CHK/CHK2 Exception. A fault that results from a range violation on the CHK or CHK2 instruction. This exception uses vector 6, a default stack frame on the 68000 and 68010, and a format 2 stack frame on the 68020 and later chips.

Chop Mode. See Round to Zero.

COBOL. COmmon Business Oriented Language. An early language designed for automating financial applications.

Common Subexpression Elimination. An optimization technique in which expressions appearing in more than one calculation are performed only once.

Condition Code Register. The 68010/020/030 allows the user program to retrieve and set the condition codes in the Status Register (SR) using special register designation. The Condition Code Register

(CCR) denotes the lower 8 bits of the Status Register. The ability for applications to read all 16 bits of the status register is one of the design flaws in the original 68000 chip.

Condition Code. One of a set of bits that contain information about the results of an instruction. The 68000 has 5 such bits contained in the Condition Code Register.

Context Switching. The act of switching programs or tasks in a multitasking system.

Coprocessor. A chip that forms an extension to the 68020 or 68030. The MMU for the 68020 (68851) and the floating point coprocessors (68881 and 68882) are the best examples of this technique. The coprocessor implements additional instructions that appear to be executed by the main processor.

Coprocessor ID. Bits 11–9 of a coprocessor instruction word. These bits indicate which coprocessor will execute the instruction. Current values are 000 (MMU) and 001 (floating point unit).

Coprocessor Protocol Exception. An exception that results from a violation of the hardware protocol between the main processor and a coprocessor. This exception uses vector 13 with a format 9 stack frame. It is not supported on the 68000 or 68010.

CPU. See Central Processing Unit.

CPU Root Pointer. (CRP). An MMU root pointer register that contains the address of the translation table for addresses generated by user mode (and optionally supervisor mode) instructions.

CRP. See CPU Root Pointer.

Current Access Level Register. An 8-bit register on the 68851 that contains the current access level.

Data Cache. A cache memory that works only on memory locations fetched as data. The 68030 has this type of cache; the 68020 does not.

Data Register. One of eight 32-bit registers in 68000 family chips. Data registers can be used as byte, word, or long quantities. The unused upper bits are not modified as they are in an address register. Data Registers are named D0–D7.

Data Register Direct. A 68000 addressing mode in which the operand is in a data register.

Debugger. A program that permits interactive control of another program. Examples include the MACSbug and SADE debuggers on Macintosh and the adb debugger on UNIX.

Demand Paging. See Virtual Memory.

Denormalized Number. A floating point number with a zero exponent and leading zero bits in the fraction. A denormalized number is too small to represent as a normalized number.

Descriptor. An entry in an MMU translation table.

Destination Function Code. A 3-bit register used both as the function code to store the destination operand on a MOVES instruction and by several of the MMU instructions described in Chapter 5.

DFC. See Destination Function Code.

Dhrystone. An often-abused benchmark for CPU integer performance, which was intended to provide a statistically correct sampling of integer operations in a high-level language (originally Ada).

Disassembly. The process of converting machine code into a human-readable (assembly language) form.

Disk. A rotating device coated with a magnetic material that stores information. The most common types are floppy disks (plastic substrate) and hard disks (metallic substrate).

Dispatcher. A program module in a multitasking system that switches between tasks.

Division By Zero. An exception that results from attempting division by zero. This exception uses vector 5 and a default format stack frame.

DMA Root Pointer. A 68851 register that contains the address of the translation table for addresses generated by external devices.

Double Precision. A floating point data format consisting of a sign bit, an 11-bit exponent, and a 52-bit fraction.

Double-Bus Fault. A situation in which a 680x0 chip references illegal memory while processing an exception. Possible causes include an incorrect value in the Vector Base Register (VBR) on a 68010 or later chip, an odd stack pointer on the 68000 or 68010, or an odd address in the vector. A double-bus fault causes the CPU to halt. Resetting the chip is the only solution.

DRP. See DMA Root Pointer.

Dyadic Operation. A floating point instruction that takes two source operands.

Dynamic Bus Sizing. The ability of the 68020 and later chips to use an 8-, 16-, or 32-bit data bus on an individual cycle basis. This feature allows connection of 8-, 16-, and 32-bit devices to the same chip with no software-visible difference (except speed).

Early Termination Descriptor. An entry in an MMU translation table that maps a region of contiguous memory. This feature is useful for simulating the 68030 Transparent Translation feature on a 68851, for instance.

EBCDIC. Extended Binary Coded Decimal Interchange Code. A method for assigning byte values to alphanumeric characters. This standard grew out of punched cards used on IBM mainframes and is still the standard used on descendants of the IBM 360/370 mainframe computer product line. EBCDIC uses the following value assignments:

Lower

		0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
	0x0	NUL	SOH	STX	ETX	HT	PF	LC	DEL			SMM	VT	FF	CR	SO	SI
	0x1	DLE	DC1	DC2	TM	RES	NL	BS	IL	CAN	EM	CC	CU1	IFS	IGS	IRS	IUS
	0x2	DS	SOS	FS		BYP	LF	ETB	ESC			SM	CU2		ENQ	ACK	BEL
U	0x3			SYN		PN	RS	UC	EOT				CU3	DC4	NAK		SUB
	0x4	SP										¢	•	<	(	+	|
p	0x5	&										!	$	*	)	;	¬
	0x6		/										,	%	_	>	?
p	0x7											:	#	@	'	=	"
	0x8		a	b	c	d	e	f	g	h	i						
e	0x9		j	k	l	m	n	o	p	q	r						
	0xA			s	t	u	v	w	x	y	z						
r	0xB																
	0xC		A	B	C	D	E	F	G	H	I						
	0xD		J	K	L	M	N	O	P	Q	R						
	0xE			S	T	U	V	W	X	Y	Z						
	0xF	0	1	2	3	4	5	6	7	8	9						

Since this standard was derived from punched cards (which are oriented toward decimal numbers), EBCDIC is biased toward codes which have 0-9 in the low four bits.

Effective Address. A 6-bit field in an instruction word (with an optional 16-bit extension word) that indicates the addressing mode for an instruction operand. This field is usually found in bits 5–0 of the first word of an instruction. (See also Addressing Modes.)

EOR. See Exclusive OR.

Exception. A feature on the 680x0 family that encompasses both external interrupts and program faults.

Exception Handler. Code that processes an exception condition.

Exclusive OR. An operation between two binary quantities, in which the result has a one bit in positions where only one of the source operands has a one bit. If both operands have a zero bit or both have a one bit, the result is zero.

Extend Bit. This bit is a copy of the Carry bit for some operations. It is used for multiple precision arithmetic.

Extended Precision. A floating point data format consisting of a sign bit, a 15-bit exponent, and a 64-bit mantissa. The memory form of this type of data also has 16 bits that are reserved.

Fault. An exception taken by the processor when a fatal error is encountered while executing an instruction. Examples include division by zero, illegal opcode, and attempting to access non-existent memory.

FC0–FC2. Three bits that indicate the type of bus cycle being attempted. These bits distinguish between supervisor and user modes, between code and data, and between normal and control cycles.

FIFO. See First In First Out.

First In First Out. (FIFO). A queuing mechanism that gives priority to the amount of time a requestor has been waiting.

Floating Point Control Register. (FPCR). A 16-bit register that contains bits to both enable floating point exceptions and control the default rounding technique.

Floating Point Data Register. One of eight extended precision registers (named FP0–FP7) used in floating point data operations. Most floating point operations (particularly those that are compute-intensive) require the destination operand to be a floating point data register.

Floating Point Instruction Address Register. (FPIAR). A 32-bit register that identifies the location of the instruction responsible for a floating point exception.

Floating Point Packed Decimal Format. A floating point data format with 3 BCD exponent digits and from 1–17 BCD digits for the mantissa.

Floating Point Status Register. (FPSR). A 32-bit register that contains information about pending and accrued exceptions, quotients from divide operations, and the floating point condition codes.

Floating Point Unit. A 68881 or 68882 coprocessor chip.

Floating Zero Divide Exception. A floating point exception that results from attempting division by zero. This exception uses vector 50 with a format 0 or format 9 stack frame.

Format Error Exception. A fault that results from a 68010 or later CPU detecting an invalid format code on an RTE or cpRESTORE type instruction. This exception uses vector 14 with a default stack frame.

FORTRAN. (FORmula TRANslation). A very old computer language invented in the late 1950s, FORTRAN is still the dominant language in numerical applications despite its age.

FPCR. See Floating Point Control Register.

FPIAR. See Floating Point Instruction Address Register.

FPSR. See Floating Point Status Register.

FPU. See Floating Point Unit.

Frame Pointer. A register (usually A6) that points to an area describing the currently executing procedure. This area usually has a pointer back to the previous area so that nested procedure calls can be unwound if desired.

Gaussian Elimination. A technique for solving a set of simultaneous linear equations. The coefficient matrix is converted to triangular form and the equations solved by back substitution.

Guard Bit. The first extra bit in the internal data format of a floating point number intended to reduce precision loss in inexact results (See Round Bit and Sticky Bit).

Hack. See Kludge.

Heap. A memory area typically used for dynamic allocation by a user program. This area is managed by functions like **new** in Pascal, and **malloc()** in C.

Hex. See Hexadecimal.

Hexadecimal. A numbering technique that has 16 digits, 0–9 and A–F. This scheme uses base 16 and is easily transformed to and from binary, which is the numbering scheme used by computer hardware.

Hidden Bit. The most significant bit of a single- or double-precision floating point number. In normalized form, this bit is always a one. Since these formats do not store it, an additional bit of precision results.

High-level Language. A programming language that requires far less detail than assembly language. Examples include BASIC, C, COBOL, FORTRAN, and Pascal.

IEEE. (Institute for Electrical and Electronic Engineers). A professional organization that functions as a clearinghouse for technical standards and guards industrial coalitions from anti-trust actions.

IEEE Floating Point. An IEEE standard for floating point arithmetic that has been widely adopted. The Intel, National Semiconductor, Motorola, and many other instruction sets now conform to this standard (ANSI-IEEE Std 754-1985).

IEEE Remainder. The remainder calculated by the FREM instruction, which always uses the round to nearest rounding mode.

Illegal Instruction. A fault that results from attempting to execute an invalid opcode. Illegal instructions use vector 4 with default format stack frame.

Immediate Mode. A 68000 addressing mode in which the operand immediately follows the instruction in memory.

Indirect Descriptor. An entry in an MMU translation table that references a page entry in another table. This is useful for memory that is shared between multiple programs.

Inexact Result Exception. A floating point exception that results from a loss of precision in a floating point operation (the INEX1 or INEX2 exception bits become set in the FPSR). This exception uses vector 49 with a format 0 or format 9 stack frame.

Infinity. A floating point data value indicating that a number is either too large to represent or mathematically impossible (for example, division by zero).

Instruction. A bit pattern that causes a CPU to perform an operation. Programs are comprised of a series of instructions that implement the desired functionality.

Instruction Cache. A cache memory that works only on memory locations fetched as instructions. The 68020 has only this type of cache.

Interrupt. The ability for an external device to cause the main processor to suspend current activity and execute another section of code.

Interrupt Mask. Bits 10–8 of the Status Register. The value in this field controls the lowest interrupt priority that the CPU will recognize. The exception is level 7, which is always recognized.

Interrupt Stack Pointer. The stack pointer used when the CPU is in Supervisor mode (Status Register bits 13–12 have the value 10). This register is called the System Stack Pointer (SSP) on the 68000 and 68010 CPUs.

Invalid Descriptor. An entry in an MMU translation table indicating that the selected address range has no valid translation. This feature is useful for virtual memory to represent memory pages that are not resident in physical memory.

ISP. See Interrupt Stack Pointer.

K. An abbreviation for "kilo". Although K usually means 1,000, in the computer world it represents 2^{10}, or 1,024.

Kludge. An inelegant solution to a difficult problem often necessitated by a scheduling requirement. The principal problem with a kludge is the lifetime of the software that hosts the kludge. All future versions of the software must remain compatible with the kludge.

LaserWriter. An Apple peripheral device capable of printing 300 dots per inch at eight pages per minute. (Drafts of this book were produced on a LaserWriter II/NTX, which has been dubbed "TreeSlayer.")

Last In First Out. (LIFO). A queuing discipline in which the last item entered into the queue is the first item taken out. The 680x0 stack feature uses this technique.

LIFO. See Last In First Out.

Line A Emulator Trap. An exception that results from attempted execution of an opcode whose first 4 bits have the value $A. This exception is used for the application to system software interface on the Macintosh. The Line A Emulator Trap uses vector 10 with a default stack frame on the 68000 and 68010 and a format 2 stack frame on the 68020 and later chips.

Line F Emulator Trap. An exception that results from attempted execution of an opcode whose first 4 bits have the value $F. On systems with coprocessors (such as the 68881 or 68851), some opcodes are valid coprocessor instructions. The Line F Emulator Trap uses vector 11 with a default stack frame on the 68000 and 68010 and a format 2 stack frame on the 68020 and later chips.

Line Feed Character. See Newline Character.

Linker. A program that takes one or more programs written in a programming language and combines them to produce a single, executable program. The MPW linker is called LINK; the UNIX linker is called ld.

Little Endian. A term that describes the ordering of bytes in memory. Little endian machines have the most-significant byte at the highest address. The 8086, PDP-11, VAX-11 and many other machines use little endian byte ordering.

Logical Address. A memory address used by a program. The logical address typically refers to the input side of an MMU.

Logical Shift. An operation on a binary number in which an operand is moved to the left or to the right. Zeros are added to fill the uppermost or lowest bit (for a right or left shift, respectively).

Long. A collection of 32 bits (4 bytes). A long can have 4,294,967,296 unique values.

Longword. See Long.

Loop Unrolling. An optimization technique in which the instructions inside a loop are replicated to allow more than one iteration to take place before the loop overhead code is executed.

M. An abbreviation for "mega." Although M usually means 1,000,000, in the computer world M means 2^{20}, or 1,048,576 (1K times 1K).

MacDraw. A graphics application distributed by Claris Software. This program was used to create the illustrations for this book.

Machine Code. Binary bit patterns directly executed by a CPU.

Master Mode. The state that occurs when a 680x0 CPU has both Status Register bits 13 and 12 set to one. The CPU can execute privileged instructions and uses the Master Stack Pointer (MSP). This mode is not implemented on the 68000 and 68010.

Master Stack Pointer. The stack pointer used when the CPU is in Master Mode (Status Register bits 13–12 have the value 11). This register is not implemented on the 68000 and 68010.

Memory. A device capable of storing and retrieving information. Computer memory devices include RAM devices, disks, magnetic tapes, and optical devices like CDs and WORM drives.

Memory Indirect Post-Indexed. A 68020 addressing mode in which the operand is in a memory location. The memory address of the operand is derived as follows: The contents of an address register are added to a 16- or 32- bit value called a base displacement. This quantity is used as a memory address to fetch an indirect pointer (32 bits). The indirect pointer is added to a 16- or 32-bit value called the outer displacement. This sum is then added to a scaled index register, and the result is the address of the operand in memory. The 16-bit values are sign-extended to 32 bits before use. The scale factor applied to the index register can be 1, 2, 4, or 8. You can individually suppress the index register, the address register, and both the displacement fields, as desired.

Memory Indirect Pre-Indexed. A 68020 addressing mode in which the operand is in a memory location. The memory address of the operand is derived as follows: The contents of the program counter register are added to a 16- or 32-bit value called a base displacement. This sum is then added to a scaled index register and this quantity is used as a memory address to fetch an indirect pointer (32 bits). The indirect pointer is added to a 16- or 32-bit value called the outer displacement. The result is the address of the operand in memory. The 16-bit values are sign-extended to 32 bits before use. The scale factor applied to the index register can be 1, 2, 4, or 8. You can individually suppress the index register, the program counter register, and both the displacement fields, as desired. The value used as the program counter is the address of the instruction plus 2.

Memory Management Unit. (MMU) A hardware device that sits between a processor and memory, translating memory addresses to provide memory protection and relocation. Sophisticated MMUs such as the 68030 and 68851 allow virtual memory support (see Virtual Memory).

Microsoft Word. A word processing application that runs on Apple Macintosh and IBM PC computer products. The Macintosh version was used to write the information you are now reading.

MMU. See Memory Management Unit.

MMU Access Violation Exception. An exception signaled by the 68851 when an attempted memory operation would violate the access level restrictions. This exception uses vector 58 with a format 2 stack frame. This exception does not occur on a 68030.

MMU Configuration Exception. An exception signaled by the MMU when the TC register is loaded with an invalid value, an invalid descriptor is loaded into a root pointer register, or a PRESTORE instruction is attempted with an invalid state frame. This exception uses vector 56 with a format 2 stack frame.

MMU Illegal Operation Exception. An exception signaled by the 68851 when a PLOAD or PTEST instruction is attempted while the MMU is disabled. This exception uses vector 57 with a format 2 stack frame. This exception does not occur on a 68030.

MMU Status Register. (MMUSR or PSR). A 16-bit MMU register that contains translation status bits for use in address validation or memory-related exception handling.

MMUSR. See MMU Status Register.

Mnemonic. An acronym or abbreviation generally used for an assembly language instruction.

Modem. A device that interfaces a computer to a telephone line. This technology allows two computers to exchange information over great distances.

Monadic Operation. A floating point instruction that takes a single source operand.

MPW. (Macintosh Programmer's Workbench). A collection of programs that allows you to write other programs on the Macintosh. Languages supported by this environment include Assembly Language, C, C++, and Pascal.

MSP. See Master Stack Pointer.

Multitasking. An operating system technique that appears to run several programs (called tasks) at once. Since the time required to switch tasks is very short, the programs appear to run simultaneously.

NAN. See Not A Number.

Natural Logarithm. A logarithm whose base is e (approximately 2.71828).

Negative Bit. Bit 3 of the Condition Code Register. This bit is set to the same value as the most significant bit of the result of an instruction. It is one for a negative number and zero for a positive number.

Newline Character. Also called line feed, this character has a value of 10 (decimal) and causes displays to advance by one line.

Nibble. Half a byte (4 bits). One hex or BCD digit occupies a nibble.

NMI. See Non-Maskable Interrupt.

No Operation. An instruction that does nothing except to take time and force synchronization.

Non-Maskable Interrupt. An interrupt that cannot be masked. On the 680x0 family, level 7 interrupts cannot be masked.

Normalized Number. A floating point number in the range 1.0 to 2.0. The most significant bit of the fractional part is a one.

Not A Number. (NAN). A floating point data value used to represent a result that has no mathematical meaning (for example, the square root of a negative number).

One's Complement. A binary operation implemented by the NOT instruction on the 680x0. Simply invert all the bits to take the one's complement.

Op Code. A binary pattern executable by a CPU.

Operand. A data item that is manipulated by an instruction.

Operand Error Exception. A floating point exception that results from an operation attempted with invalid data (for example, dividing zero by zero). This exception uses vector 52 with a format 0 or format 9 stack frame.

Operating System. Software that runs other software. Examples include the Macintosh O/S, IBM PC-DOS, IBM OS/2, and the AT&T UNIX systems.

OPERR. See Operand Error Exception.

OR. An operation between two binary quantities in which the result has a one bit in positions where either of the two source operands has a one bit.

Outer Displacement. A field in 68020 addressing modes that can be either 16 or 32 bits. This field is the second displacement field in memory.

Overflow (V) Bit. Bit 1 of the Condition Code register. This bit is set when an overflow occurs in an operation.

Overflow Exception. A floating point exception that indicates that the result of an operation was too large. This exception uses vector 53 with a format 0 or format 9 stack frame.

OVFL. See Overflow Exception.

Page. A unit of memory which is the smallest area transferred from disk (or a network) to and from main memory. Pages are loaded on demand in a virtual memory system.

Page Descriptor. An entry in an MMU translation table that describes a page of memory mapped by the MMU.

Page Table. See Translation Table.

Parallelism. A hardware feature that allows individual components of a system to work on different operations simultaneously. The best examples of the 680x0 family are the 68881 and 68882 coprocessors, which can work independently from the main CPU for short durations.

Pascal. A computer language originally developed for teaching programming by Niklaus Wirth of ETH Zurich. This language later became the basis for software development for the original Macintosh computer.

PC. See Program Counter.

pcc. The UNIX system Portable C Compiler. This is the C compiler used with A/UX in this book.

PCSR. See PMMU Cache Status Register.

PC Relative. A special addressing mode used for the Bcc (Branch Conditionally) and BSR (Branch to Subroutine) instructions. An immediate quantity is added to the PC to form a new address for the next instruction.

Physical Address. A memory address presented to memory typically used to refer to the output side of an MMU.

Pixel. A single point on a graphics display device.

PMMU Cache Status Register. (PCSR). A 68851 16-bit register with status information from the address translation cache.

Pop. Removing an item from the top of a stack. Pop operations on the 680x0 family use the Address Register Indirect with Postincrement Addressing mode.

Privilege Violation. An exception resulting from the attempted execution of a privileged instruction while the CPU is in user mode. This exception uses vector 8 with a default stack frame on the 68000 and 68010 and a format 2 stack frame on the 68020 and later chips.

Privileged Instruction. An instruction that may not be executed in user mode. These are usually instructions that can compromise the function of a multitasking system if executed by a user program. Examples include RTE, ANDI to SR, RESET, and STOP.

Program Counter. 680x0 chips contain a 32-bit register that specifies the address of the next instruction to be executed. This register is called the program counter, or PC.

Program Counter Indirect with Displacement. A 68000 addressing mode in which the operand is in a memory location whose address is specified by the program counter register plus a 16-bit sign- extended displacement. The value used as the program counter is the address of the instruction plus 2.

Program Counter Indirect with Index. (8-bit Displacement). A 68000 addressing mode in which the operand is in a memory location whose address is specified by the program counter register plus an index register plus a sign-extended 8-bit displacement. The index register may be either an address or data register. You can use either the low-order 16 bits or all 32 bits of the index register. On 32-bit CPUs, the contents of the index register can be multiplied by 1, 2, 4, or 8. This scaling feature is not implemented on the 68000 or 68010. The value used as the program counter is the address of the instruction plus 2.

Program Counter Indirect with Index. (Base Displacement). A 68020 addressing mode in which the operand is in a memory location. The address of the operand is calculated as the contents of the program counter register plus the contents of a scaled index register plus a 16- or 32-bit displacement (16-bit displacements are sign-extended to 32 bits before use). The index register contents can be scaled by 1, 2, 4, or 8 before use (the index register itself is not modified). This mode is not implemented on the 68000 or 68010. The value used as the program counter is the address of the instruction plus 2.

Program Counter Memory Indirect Post-Indexed. A 68020 addressing mode in which the operand is in a memory location. The memory address of the operand is derived as follows: The contents of the program counter register are added to a 16- or 32-bit value called a base displacement. This quantity is used as a memory address to fetch an indirect pointer (32 bits). The indirect pointer is added to a 16- or 32-bit value called the outer displacement. This sum is then added to a scaled index register, and the result is the address of the operand in memory. The 16-bit values are sign-extended to 32 bits before use. The scale factor applied to the index register can be 1, 2, 4, or 8. You can individually suppress the index register, the program counter register, and both the displacement fields, as desired. The value used as the program counter is the address of the instruction plus 2.

PSR. See MMU Status Register.

Push. Adding an item to the top of a stack. Push operations on the 680x0 family use the Address Register Indirect with Predecrement Addressing mode.

Qsort. A famous recursive sort algorithm invented by C. A. R. Hoare.

RAM. (Random Access Memory). A memory device consisting of silicon logic elements that store information.

Read/Modify/Write Cycle. A type of bus cycle used in systems with more than one active entity on the bus. This type of cycle (implemented using a Test And Set instruction on the 680x0) fetches, updates, and stores data on the bus without allowing any other entity to access the data. This is useful for setting flag values for synchronizing non-shareable resources in a computer system, for instance.

Recursion. A recursive routine is one which is defined in terms of itself. The qsort program presented in Chapter 6 is an example of this technique.

Reentrant Code. On a multitasking system, code that can be executed by more than one task at a time.

ROM. Read Only Memory.

Root Pointer. An MMU register that points to the beginning of the translation table in memory.

Root Pointer Table. A 68851 feature that caches the eight most recently used values of the CPU root pointer.

Rotate. A binary operation that shifts all bits in an operand in one direction or the other. Bits taken off the end are inserted in the other end, possibly through an extra bit (the X bit on the 680x0).

Round Bit. The second extra bit in the internal floating point data format (see Guard Bit and Sticky Bit). This bit indicates how rounding is to proceed.

Round to Minus Infinity. A floating point rounding mode that chooses the value closest to minus infinity.

Round to Nearest. A floating point rounding technique that chooses the value closest to the infinitely precise result.

Round to Positive Infinity. A floating point rounding mode that chooses the value closest to positive infinity.

Round to Zero. Also called "Chop Mode," this floating point rounding technique truncates any bits to the right of the rounding point. This mode is used by the FORTRAN language.

Scatter/Gather I/O. The ability of an MMU to translate memory addresses presented by external devices. The 68851 has this capability; the 68030 MMU does not.

SCC. See Stack Change Control Register.

SFC. See Source Function Code.

Sign Extension. When moving a smaller data item to a larger one, the upper bit of the source is copied to the extra bits in the destination. For example, when copying the byte quantity $F0 to a word, the result is $FFF0.

Signaling NAN Exception. A floating point exception that results from attempted use of a signaling NAN as one of the operands of an instruction. This exception uses vector 54 with a format 0 or format 9 stack frame.

Signaling NAN. A floating point data value that indicates a mathematically meaningless result (for example, the square root of -1) and that signals an exception when referenced.

Simultaneous Linear Equations. A series of at least n equations with n variables.

Single Precision. A floating point data format consisting of a sign bit, an 8-bit exponent, and a 23-bit fraction (roughly 7 decimal digits of accuracy).

SNAN. See Signaling NAN.

Source Function Code. A 3-bit register used as the function code to fetch the source operand on a MOVES instruction. Also used by several of the MMU instructions described in Chapter 5.

Spurious Interrupt. An interrupt resulting from an interrupt request that did not supply an interrupt vector (a hardware protocol violation). This interrupt uses vector 24 with a default stack frame.

SR. See Status Register.

SRP. See Supervisor Root Pointer.

SSP. See Interrupt Stack Pointer.

Stack. A temporary memory area that is implemented using a register. The register indicates the address of the top item in the stack and moves to lower or higher addresses to add or remove (respectively) data from the stack. A stack is a Last In First Out (LIFO) data structure. On the 680x0 family, address register A7 is used as the stack pointer.

Stack Change Control Register. A 68851 8-bit register used to govern whether a stack change is imposed when changing from one access level to another.

Stack Frame. A data structure pushed onto the stack as a result of an exception.

Stack Overflow. Attempting to add an item to a stack that is already full.

Stack Pointer. A register that contains the address of the top item in a stack. On the 680x0 family, address register A7 performs this function for the 68000 hardware stack.

Stack Underflow. Attempting to pop more items than were pushed on a stack.

Status Register. A 16-bit register that contains the 680x0 condition codes, interrupt mask, and mode control bits (supervisor/master/user and trace).

Sticky Bit. A third extra bit in the internal floating point data format (see Guard Bit and Round Bit). This bit is non-zero if any bits to the right of the representable number are non-zero.

Subroutine. A sequence of code that implements an operation that is required at multiple points in a program (or in a series of programs). The 68000 supports instructions (JSR, BSR, and RTS) that call and return from a subroutine.

Supervisor Mode. The state that occurs when a 680x0 CPU has Status Register bit 13 and 12 set to 10. The CPU can execute privileged instructions in this state and uses the Interrupt Stack Pointer (ISP).

Supervisor Root Pointer. (SRP). An MMU root pointer register that, if enabled, contains the address of the translation table for addresses generated by the CPU in supervisor mode.

System Stack Pointer. See Interrupt Stack Pointer.

Table Descriptor. An entry in an MMU translation table that points to another translation table.

Task. A separately executing program entity, usually in a system where many of these entities execute together (see Multitasking). Each task has its own set of registers. A software module called a dispatcher switches between tasks, according to some scheduling discipline.

TC. See Translation Control Register.

TLB. See Address Translation Cache.

Trace Bit. Bits 15 and 14 in the status register (Bit 14 is not present on the 68000 and 68010). Setting one of these bits will cause a trace exception after either the next instruction (bit 15) or the next branch instruction (bit 14).

Trace Trap. An exception that results from executing an instruction with one of the Status Register Trace bits (15 and 14) set to one. This exception uses vector 9 with a default stack frame on the 68000 and 68010 and a format 2 stack frame on the 68020 and later chips.

Translation Control Register. A 32-bit MMU register that contains the master enable, Supervisor Root Pointer enable, Function Code Lookup enable, and the number of address bits to allocate for each level of the translation table.

Translation Lookaside Buffer. See Address Translation Cache.

Translation Table. The address translation table used in an MMU with an Address Translation Cache (also called a Page Table).

Transparent Translation. A 68030 MMU feature that allows mapping of contiguous 16Mb memory address spaces, with very little translation overhead. There are two such regions mapped by registers TT0 and TT1.

Trap Instruction Exception. An exception that results from the execution of a TRAP #n instruction. There are sixteen such instructions (TRAP #0 through TRAP #15), each generating a unique exception. The exceptions use vectors 32–47 with default stack formats.

TRAPV/TRAPcc Exception. An exception resulting from a TRAPV or TRAPcc instruction. This exception uses vector 7 with a default stack frame on the 68000 and 68010 and a format 2 stack frame on the 68020 and later chips.

Triangular Form. A matrix in which all elements below the diagonal equal zero. A linear equation coefficient matrix in this form can be solved using the back substitution technique.

Two's Complement. A binary representation that allows both negative and positive numbers to use the same instructions. A number is made negative by inverting all the bits and adding one.

Underflow Exception. A floating point exception indicating that the result of an operation was too small. This exception uses vector 51 with a format 0 or format 9 stack frame.

UNFL. See Underflow Exception.

Uninitialized Interrupt. A default vector value reserved for use by peripheral chips whose vector has not been initialized by software. This exception uses vector 15 with a default stack frame.

UNIX. A computer operating system originally developed by Ken Thompson and Dennis Ritchie at Bell Laboratories. This system (currently called System V) is the basis for the A/UX system used in Appendix A.

Unnormalized Number. A floating point number with leading zeros in the fractional part and a non-zero exponent. The number can be converted to a normalized number.

Unordered Condition. The result of a floating point compare where one of the two operands was a NAN. This renders the comparison meaningless.

USENET. A loosely-organized network of UNIX and other computers that communicate mostly over dial-up telephone connections.

User Mode. The state that occurs when a 680x0 CPU has Status Register bits 13 and 12 both equal to zero. The CPU cannot execute privileged instructions and uses the User Stack Pointer (USP).

User Stack Pointer. The stack pointer used when the CPU is in User Mode (Status Register bits 13–12 have the value 00).

USP. See User Stack Pointer.

VAL. See Validate Access Level Register.

Validate Access Level Register. An 8-bit register on the 68851 containing the previous access level. This register is used to validate addresses passed by a less-privileged routine to a more-privileged routine.

VBR. See Vector Base Register.

Vector Base Register. A 32-bit register that defines the beginning of the interrupt vector area.

Virtual Memory. A software technique that keeps only a subset of a program (sometimes called the working set) in memory at a given time. Virtual memory is also called demand paging. Memory is divided into a set of small areas called pages that are brought in when needed from disk or over a network. This technique requires a hardware assist such as that of the 68851 and 68030. Virtual memory causes an exception when the program accesses a page not in memory. This technique is discussed in Chapter 5.

Von Neumann Bottleneck. A term that describes the fact that most computers are limited by the speed of the processor to memory interface since both instructions and data need to traverse this path.

Wild Branch. A programming error that is particularly difficult to locate. The program winds up somewhere that is "impossible" (usually attempting to execute data rather than instructions), and you have no indication why. This situation is usually caused by an RTS or RTE instruction in which the stack pointer does not point to the correct data.

Word. A collection of 16 bits (2 bytes). A word can have 65,536 unique values.

Working Set. A term used to describe the set of pages used by a program at any given time. This is usually less than the total number of pages in the entire program, giving rise to a technique called virtual memory (see Virtual Memory).

WORM. (Write Once, Read Mostly). A WORM drive is an optical drive that can write information one time at each place on the disk.

Write Allocation. A feature of the 68030 data cache that causes a cache entry to be created when a memory location not in cache is written.

Z8000. A 16-bit CPU chip made by Zilog. This chip's chief claim to fame is the System 8000. Large quantities of this machine (and therefore the chip) were purchased by the IRS.

Zero Bit. Bit 2 of the Condition Code Register. This bit is set to the complement of the OR of all bits in the result of an instruction. It is set only if all bits of the result are not set.

INDEX

| logical OR operator, 356
$ hexadecimal convention, 4, 69
& logical AND operator, 356
() logical notation, 356
- specify range of registers with, 237, 470
/ register names specified by, 237, 467, 470
_ underscore, 633
~ logical NOT operator, 356

68000 family
    assembly language syntax, 2-5
    evolution, *7*-10
    faults, 46-58
    interrupts, 58-61
68000 microchip (CPU), 8
    addressing modes, 18-27
    instruction trace routine, 680-83
    RTE instruction format, 298-99
68010 microchip (CPU), 8
    instruction tracer routine, 677-80
    RTE instruction format, 298-99
68020 microchip (CPU), 9
    addressing modes, 27-36
    cache memory, 37-39
    data types supported by, 10-12
    instruction tracer routine, 677-80
    registers, *12*
    RTE instruction, 298-99
    RTM instruction, 300-301
68030 microchip (CPU), 10
    addressing modes, 27-36
    cache memory, 39-44
    exceptions, 568-70
    instruction tracer routine, 677-80
    MMU architecture, 544
    register set, *546*-51
    RTE instruction, 298-99
    transparent translation registers, 553-54
68451 microchip (MMU), 543
68851 microchip (MMU), 61
    access control registers, 555-57
    architecture, 543, 544
    breakpoint registers, 554-55
    cache status register, 557
    exceptions, 568-70
    register set, *546*-52
    root pointer table, 552-53
68881 microchip (floating point), 61, 343
    concurrency, 374-75
    coprocessor detection, 379
    register set, 350, *351*
    rounding modes supported by, 347-48
    state frames, *361*
68882 microchip (floating point), 61, 343
    concurrency, 375-78
    full concurrency, 377
    minimal concurrency, 375-76
    no concurrency, 375
    partial concurrency, 376
    coprocessor detection, 379
    performance optimization, 377-78
    register set, 350, *351*
    rounding modes supported by, 347-48
    state frames, *361*

ABCD instruction (ADD Binary Coded Decimal with extend), 70-71
Absolute addressing, 3-4, 35-36, 63
Absolute Long Address addressing mode, 35-*36*

Absolute Short Address addressing mode, 35
Access Control (AC) register, *556*-57, 610
Accrued Exception byte (FPSR), *355*
ADDA instruction (ADD Address), 74-75
ADDI instruction (ADD Immediate), 76-77, 78
ADD instruction (ADD Binary), 72-73, 81
Addition
  add effective address operand to an address register, 74-75
  add immediate data to effective address operand, 76-77
  add immediate operand to effective address operand, 78-79
  add operand to floating point data register, 392-94
  add two 8-bit BCD numbers, 70-71
  add with extend conditions, 80
  two's complement, 72-73
ADDQ instruction (ADD Quick), 78-79
Address. *See* Logical address; Physical address
Addressing Error exception, 48
Addressing modes, 3-4, 18-36
  instruction set description of, 67-68
  summary, 18(table)
Addressing modes, floating point, 382
Addressing modes, Memory Management Unit, 571-72
Address register(s), 13
  add effective address operand to, 74-75
  compare, with effective address operand, 156-58
  compare memory locations with contents of, 162-63
  exchange contents of, 200-201
  floating point, 356
  load effective address into, 211-12
  move data from an effective address operand to, 223-24
  push, onto stack, 213-14
  subtract effective address operand from, 314-15
  zero suppressed, 25, 27
Address Register Direct addressing mode, 19, 62-63
Address Register Indirect addressing mode, 19-20
Address Register Indirect with Displacement addressing mode, 21-22, 28
Address Register Indirect with Index (8-bit Displacement) addressing mode, 22-23
Address Register Indirect with Index (Base Displacement) addressing mode, 23-25
Address Register Indirect with Postincrement addressing mode, 20
Address Register Indirect with Predecrement addressing mode, 21
  add two bytes in memory using, 70-71
  add two memory locations using, 80-81
  subtract two bytes in memory using, 305-6
Address space
  linear vs. segmented, 8
  move, 246-48
Address Translation Cache (ATC), 541, *542*
  flush entries from, with PFLUSH, 581-84
  flush entries from RPT and, with PFLUSHR, 585-87
  indicate internal status of, with Cache Status register, 557
  load entries, 588-90
ADDX instruction (ADD Binary with eXtend), 80-81
ANDI instruction (AND Immediate), 84-85
AND instruction (AND Logical), 82-83
ANDI to CCR instruction (AND Immediate to Condition Code Register), 86-87
ANDI to SR instruction-Privileged (AND Immediate to Status Register), 88-89
AND logical operator, 82
  on data register and effective address operand, 82-83
  on effective address operand using immediate data, 84-85
  on immediate operand and Condition Code Register, 86-87
  on immediate operand and Status Register, 88-89
AND logical operator, floating point, 356, 362
Apple MPW C language compiler, assembly
  interface with data types, 614-16
  global data access, 620-21
  procedure calling standard, 616-20

APU arithmetic processor, 375, 377

Arc cosine function, 389-91

Architecture, 1-2, 7-64
  68000 family evolution, 7-10
  addressing modes, 18-36
  coprocessor operations, 61-64
  data types, 10-12
  exception processing, 44-58
  floating point unit *(See* Floating point unit (FPU) architecture)
  interrupt, 58-61
  memory cache operation, 36-44
  memory management unit *(See* Memory Management Unit architecture)
  register set, 12-17

Arc sine function, 395-97

Arc tangent function, 398-400

Arithmetic operations, monadic and dyadic floating point, 374, 376, 381

Arithmetic shifts
  left, 90-92
  right, 93-95

ASCII digits
  convert BCD into, 339-41
  convert into BCD, 278-79

ASL instruction (Arithmetic Shift Left), 90-92

ASR instruction (Arithmetic Shift Right), 93-95

Assembly language programming, 613-84
  advantages of, 613-14
  conclusions on, 683-84
  interfacing to high-level languages, 614-26
    Apple MPW C language, 614-21
    Macintosh Pascal, 623-25
    UNIX pcc compiler, 621-23
  notational conventions, 2-5
  organization of this book, 1-2
  performance improvements, 626-27
  programming examples, 627-84
    C string functions, 627-32
    floating point, 661-73
    graphics, 652-61
    instruction tracer, 676-83
    loop unrolling, 632-40
    scaling, 673-76
    searching, 648-52
    sorting, 640-48

Atari ST series, 2, 3

Autovectored interrupts, 60

Autovectoring technique, 58-59

Back substitution, 667

Base 2 logarithm, 443-45

Base 10 logarithm, 440-42

Base/bounds memory management, 538, 539-41
  heap, 540
  program memory areas for, *539*
  segmented, *540*

Base e
  calculate logarithm with, 446-48
  calculate logarithm (x+1) with, 449-51
  e^x function, 422-24
  e^x-1 function, 425-27

Base Suppress (BS) flag, 25, 26, 31, 33

Bcc instruction (Branch conditionally), 96-98, 131, 404, 565
  conditions and meanings, 97-98(table)

BCD. *See* Binary-Coded Decimal (BCD) data type; Binary-Coded Decimal (BCD) numbers

BCHG instruction (Test a Bit and Change/Invert), 99-100

BCLR instruction (Test a Bit and Clear), 101-2

BFCHG instruction (Test Bit Field and Change), 103-5

BFCLR instruction (Bit Field Clear), 106-8

BFEXTS instruction (Bit Field Extract Signed), 109-11

BFEXTU instruction (Bit Field Extract Unsigned), 112-14

BFFFO instruction (Bit Field Find First One), 115-17

BFINS instruction (Bit Field Insert), 118-20

BFSET instruction (Bit Field Set), 121-23

BFTST instruction (Bit Field Test), 124-26

Big endian byte order, 11, 305, 321

Binary-Coded Decimal (BCD) data type
  conversion, 9
  definition and layout, 11, *12*

Binary-Coded Decimal (BCD) numbers
  add two 8-bit numbers, 70-71
  invert sign of, 259-61
  pack (convert ASCII and EBCDIC into), 278-79
  representation of infinity by, *349*
  representation of Not-A-Number by, *350*
  subtract, 305-6
  unpack (convert BCD into ASCII and EBCDIC), 339-41
Binary floating point data formats, 344-46
Binary floating point number, 344-46
Binary form of instructions, 68
Binary number
  one's complement, 268-69
  two's complement
    add, 72-73
    invert sign of, 262-63, 264-65
    subtract, 312-13
  representation of infinity by, *348*
  representation of Not-A-Number by, *350*
bisect() function, 662
Bisection method of solving equations, 661-65
Bit
  find most significant, 115-17
  guard, 347
  hidden, 345
  integer, 345
  round, 247
  sticky, 347
  test and change, 99-100
  test and clear, 101-2
  test and set, 129-30
  test and set condition codes register according to, 133-34
Bit data type, defined, 11
Bit field
  clear, 106-8
  copy, into a data register, 109-11
  find most significant bit in, 115-17
  insert, 118-20
  set, at the designated effective address to all ones, 121-23
  test and change, 103-5
  test and set condition codes, 124-26
Bit field data type, 9
  definition and layout, *11*
BKPT instruction (Breakpoint), 127-28
BRA instruction, 207
Branch(es)
  conditional, 96-98
  on coprocessor condition, 167, 168
  on floating point condition, 404-6
  on MMU condition, 575-77
  to subroutine, 131-32
  test coprocessor condition, decrement, and, 168
  test floating point condition, decrement, and, 416-18
  test main processor condition, decrement, and, 177-79
  test MMU condition, decrement, and, 578-80
Branch/Set on Unordered (BSUN) enable bit (FPCR), 351, 351-52, 358. *See also* Unordered condition (BSUN) exception
Breakpoint Acknowledge Control Register (BACRx), *554*-55
Breakpoint Acknowledge cycle, 53-54
  cause with BKPT instruction, 127-28
Breakpoint Acknowledge Data Register (BADRx), *554*-55
Breakpoint Enabled Format State Frame, *568*
Breakpoint registers, MMU, *554*-55
Bresenham circle algorithm, 657-61
Bresenham line algorithm, 653-57
bsearch optimization, 648-52
BSET instruction (Test a Bit and Set), 129-30
BSR instruction (Branch to Subroutine), 131-32, 209, 295, 616
  reverse with RTS instruction, 304
BTST instruction (Test a Bit), 133-34
Burst mode technique, 39, 43, 44, 266
  fetch order, *39*
Bus
  dynamic bus sizing, 241
  error processing, 47
  generate arbitrary bus cycle, 246
  read/modify/write cycle, 141
BUSERR/Addressing Error Stack Frame, 48-53
  short and long stack frames, 50, *51*, 52, 53

  - Special Status register
    - 68010 chip, *49*
    - 68020/68030 chip, *52*
- BUSERR exception, 47
  - avoid, when moving registers to/from memory, 240
  - breakpoint and, 127
  - MMU-generated, 569
  - spurious interrupt and, 60
- Bus Interface Unit (BIU) flag, 362, *363*
- BUSY State Frame, 360, 364
- Byte(s), 10
  - add two bytes in memory with ABCD instruction, 70-71
  - copy, from one area to another, 632-34
  - defined, 11
  - floating point, 344
  - set according to coprocessor condition, 174-75
  - set according to CPU condition codes, 307-9
  - set according to MMU conditions, 600-602
  - subtract, 305-6

- CAche Address Register (CAAR), 17, 39, 42
- CAche Control Register (CACR), 17, 613
  - 68020 chip, *38*
  - 68030 chip, *42*-44
- Cache hit, 38
- Cache lines, 39
- Cache memory, 9, 36-44. *See also* Address Translation Cache (ATC)
  - 68020 cache, 37-39
    - architecture, *37*
    - cache address register, *39*
    - cache control register, *38*
    - cache hit, 38
  - 68030 cache, 39-40
    - burst mode, 39-40,
    - 266 cache control, 42-44
    - cache lines, 39
    - data cache, *41*-42
    - instruction cache, *40*
    - write allocation, 41-42
    - write-through operation, 41
  - instruction cache vs. data cache, 37
  - tag array and data array in, 37-38
- Cache Status register, *557*
- CALLM instruction (CALL module), 135-40, 300
  - access control registers supporting, 555-57
  - example, 139-40 fields, 136-37
  - instruction format, *136*
  - Module Call Stack Frame format, *137*
  - module descriptor
    - layout, *136*
    - types, 138-39
  - valid addressing modes, *135*
- CAS2 instruction (Compare and Swap two operands), 144-45
- CAS instruction (Compare and Swap one operand), 141-43
- Catastrophic overflow, 373
- Catastrophic underflow, 372
- Central processing unit (CPU) architecture. *See* Architecture
- CHK/CHK2 exception, 54-55
- CHK2 instruction (Check Register Against Bounds-two operands), 148-50, 164
- CHK instruction (Check Register Against Bounds-one operand), 54, 146-47
- Chop mode (Round to Zero), 347, 437, 491
- Circle algorithm, 657-61
- circle_points() function, 658, 660
- C language
  - assembly language interface with, 614-26
    - Apple MPW C, 614-21
    - UNIX pcc Compiler, 621-23
  - bisection method of solving an equation optimized, 661-65
  - search routine optimized, 648-52
  - simultaneous linear equation solutions optimized, 665-73
  - sort routine optimized, 640-48
  - string functions optimized, 627-32
- Clear cache bit,
- CACR, 38
- Clear Data Cache Entry flag, CACR, 43
- Clear Data Cache flag, CACR, 43
- Clear Entry bit, CACR, 39
- Clear Instruction Cache bit, CACR, 44

Clear Instruction Cache Entry, CACR, 44

CLR instruction (Clear/Zero an Operand), 151-52

CMP2 instruction (Compare Register Against Bounds), 164-66

CMPA instruction (Compare with Address Register), 156-58

CMPI instruction (Compare Immediate), 159-61, 335

CMP instruction (Compare with Data Register), 153-55

CMPM instruction (Compare Memory), 162-63

Code area, MMU, 538, 539

Command Interface Register (CIR), 364

Command word, pass to coprocessor, 169

Commodore Amiga series, 2, 3

Communications of the ACM, 632

Comparison routine, 637-39

Concurrency considerations, 373-78

Condition(s), floating point, 356-60. *See also* Condition codes(s), floating point
  branch on, 404-6
  conditions described, 358-60
  conditions listed, 357(table)
  evaluate, 494-96
  IEEE aware and IEEE non-aware exceptions, 358
  no concurrency capability in instructions for, 375
  test, then decrement and branch, 416-18
  trap on, 528-30

Conditional branch, 96-98
  floating point, 404-6, 416-18
  MMU, 575-80

Conditional operations, Memory Management Unit, 565-66
  branch on MMU condition, 575-77
  set byte according to MMU condition, 600-602
  set condition codes in MMUSR, 603-6
  test MMU condition, decrement, and branch, 578-80
  trap on MMU condition, 607-9

Condition code(s), 14. *See also* Condition Code Register (CCR)
  instruction description for, 66-67
  return and restore, 302-3
  set, based on value of effective address operand, 335-36
  set, with CMP instruction, 153-55
  set, with CMPA instruction, 156-58
  set, with CMPI instruction, 159-61
  set bytes according to, 307-9
  X and Z bit, 259

Condition code(s), floating point, 356-60
  compare and set, 407-9
  not-a-number quantities, 356, 358
  notation, 356
  test and set, 531-33

Condition Code Byte (FPSR), *353*

Condition Code Register (CCR), 12, *14*
  AND operation on immediate operand and, 86-87
  conditional branch based on settings in, 96-98
  move data from, to effective address operand, 225-26
  move data to, from effective address operand, 227-28
  OR operations on, 196-97, 274-75
  test a bit and set, 133-34

config() routine optimization, 674-76

Constants, move to floating point data register, 464-66

Context switching, 378, 485-87, 488-90, 557

Control registers
  cache control *(See* CAche Control Register (CACR))
  fetch and set with MOVEC, 235-36

Control registers, floating point, 350-53
  move data between effective address and, 455-57
  move to/from effective address, 467-69

Coprocessor(s), 1, 9, 61-64
  architecture *(see* Floating point unit (FPU) architecture)
  detection, 379
  exceptions generated by, 63-64
  instruction
    description, 379-85
    format, *62*-63, *383*
  pass command word to, 169

  restore state, 170-71, 485-87
  save state, 172-73, 488-90

Coprocessor condition(s)
  branch on, 167
  set on, 174-75 test,
  then decrement and branch, 168
  TRAP on, 176

Coprocessor-Detected faults, 58

Coprocessor Protocol exception, 56-57
  floating point, 365

Copy routine, unrolled version of, 634-36

Cosine function, 410-12
  simultaneous with sine function, 506-9

cpBcc instruction (Branch on Coprocessor Condition), 63, 167

cpDBcc instruction (Test Coprocessor Condition, Decrement, and Branch), 63, 168

cpGEN instruction (Pass Command Word to Coprocessor), 63, 169

cpRESTORE instruction (Privileged), 63, 170-71

cpSAVE instruction (Privileged), 63, 170, 172-73

cpScc instruction (Set on Coprocessor Condition), 63, 174-75

cpTRAPcc instruction (TRAPP on Coprocessor Condition), 63, 176

CPU Root Pointer (CRP), 550, 551-52, 585

Current Access Level (CAL) register, *555*, 569

Data
  accessing global (*See* Global data access)
  cache memory, 37
  copy, from an effective address operand to address register, 223-24
  immediate (*See* Immediate data)
  move, from Condition Code Register to effective address, 225-26
  move, from effective address operand to an address register, 223-24
  move, to Condition Code Register from effective address, 227-28
  move between effective addresses, 221-22
  move peripheral data, 241-43

DaTa ACKnowledge (DTACK) signal, 47

Data area, MMU, 538, 539

Data Burst Enable flag, CACR, 43

Data cache
  68030 chip, *41*-42
  vs. instruction cache, 37

Data Input/Output Buffers, 52

Data register(s), 12, 13
  add, to another, 70-71
  add, to another with extended conditions, 80-81
  add, to effect address operand, 72-73
  AND operation on, and effective address operand, 82-83
  arithmetic shift left, 90-92
  arithmetic shift right, 93-95
  check against bounds (one operand), 146-47
  check against bounds (two operands), 148-50
  compare, with effective address operand, 153-55
  compare against bounds, 164-66
  copy signed bit field to, 109-11
  copy unsigned bit field to, 112-14
  exchange contents of, 200-201
  logical rotate operations on, 283, 286, 289, 292
  logical shift operations on, 215, 218
  move immediate quantity into, 244-45
  OR operation on effective address operand and, 191, 270
  sign extension operations performed on, 202-3, 204-5
  swap bits of, 324 subtract, between effective address operand and, 312-13
  subtract, one from another, 305-6, 321-23

Data register, floating point, 373
  absolute value of a number placed in, 386-88
  add operand to, 392-94
  arc cosine of a number placed in, 389-91
  arc sine of a number placed in, 395-97
  arc tangent of a number placed in, 398-400
  cosine of a number placed in, 410-12
  constants placed in, 464-66
  divide, by source operand, 419-21, 497-99
  hyperbolic arc tangent of a number placed in, 401-3
  hyperbolic cosine of a number placed in, 413-15

hyperbolic sine of a number placed in, 510-12
hyperbolic tangent of a number placed in, 522-24
move data between effective address and, 458-63
move to/from effective address, 470-73
place source operand's negative in, 477-79
set condition codes by subtracting source operand from, 407-9
simultaneous sine and cosine of a number placed in, 506-9
sine of a number placed in, 503-5
source operand's square root placed in, 513-15
subtract source operand from, 516-18
tangent of a number placed in, 519-21

Data register direct addressing mode, 19

Data types
  Apple MPW C language, 614-16
  floating point, 344-50
  Macintosh Pascal, 623-24
  supported by 68020 chip, 10-12
  UNIX pcc compiler, 622

DBcc instruction (Test Condition, Decrement, and Branch), 177-79, 416, 565
  conditions, 178-79

Debugging software
  Illegal Instruction exception use in, 53
  MACSbug, 619
  symbolic debugger output, description, 69
  Trace Trap use in, 55
  using NOP instructions in, 266-67

Decrement
  test coprocessor condition and, 168
  test floating point condition and, 416-18
  test main processor condition and, 177-79
  test MMU condition, and, 578-80

Default, exception stack frame, *45*

Denormalized binary floating point numbers, 345

Destination Function Code (DFC) register, 17, 246

Destination operand
  divide by source operand, 452-54
  floating point, 373-74
  invert bits in, 268-69
  multiply with source operand, 474-76, 500-502
  subtract from zero, 259, 262, 264

Dhrystone benchmark, 627
  effect of C string optimization program on, 632

Directed rounding modes, 348

Divide by Zero (DZ) enable bit (FPCR), 352

Division
  32-bit quantity by 16-bit quantity
    signed, 180-81
    unsigned, 185-87
  32-bit signed, 182-84
  32-bit unsigned, 188-90
  destination by source operand, 452-54
  floating point, 419-21, 497-99

Division by Zero (DZ) exception, 54, 352, 367-68
  continuing from, 371

DIVS instruction (Signed Divide-Word Form), 54, 180-81

DIVSL/DIVS.L instruction (Signed Divide-Long Forms), 54, 182-84

DIVU instruction (Unsigned Divide-Word Form), 54, 185-87

DIVUL/DIVU.L instructions (Unsigned Divide-Long Forms), 54, 188-90

DMA Root pointer DRP, 550-52

Double precision format, 344, *345*, 346

Dyadic arithmetic operations, 374, 376, 381

Dynamic bus sizing, 241

e. *See* Base e

Early termination descriptors, 558, *563*-65

EBCDIC digits
  convert BCD into, 339-41
  convert into BCD, 278-79

Effective address operand
  add, to address register, 74-75
  add, to data register, 72-73
  add immediate data to, 76-77
  add small immediate operand to, 78-79
  AND operation on, and data register, 82-83
  AND operation on, using immediate data, 84-85

  arithmetic shift left, 90-92
  arithmetic shift right, 93-95
  compare, with address register, 156-58
  compare, with data register, 153-55
  compare, with immediate data, 159-61
  copy Status Register into, 229-30
  jump to, 207-8
  jump to subroutine at, 209-10
  load, 211-12
  logical rotate operations on, 283, 286, 289, 292
  logical shift operations on, 215, 218
  move data between, 221-22
  move data between control register and, 455-57
  move data between floating point data register and, 458-63
  move data from, into Status Register, 231-32
  move data from, to Condition Code Register, 227-28
  move data to, from Condition Code Register, 225-26
  move data to address register from, 223
  move floating point control registers to/from, 467-69
  move floating point data registers to/from, 470-73
  OR operation on, using immediate data, 193, 272
  OR operation on data register and, 191, 270
  push, onto the stack, 280-81
  set condition codes based on value of, 335-36
  set to zero, 151-52
  subtract, from address register, 314-15
  subtract immediate data from, 316-17, 318-20
  subtract, between data register and, 312-13

EORI instruction (Exclusive OR Immediate), 193-95

EOR instruction (Exclusive OR Logical), 191-92

EORI to CCR instruction (Exclusive or Immediate to Condition Code Register), 196-97

EORI to SR instruction-Privileged (Exclusive OR Immediate to Status Register), 198-99

e^x-1 function, 425-27

Exception(s), 44-68
  coprocessor-generated, 63-64
  defined, 44
  exception stack frames, 45-46, 298-99
  faults, 45, 46-58
  instruction trace, 676-83
  interrupts, 58-61
  return from, 297-99
  Trap, 57-58, 328-29
  Trap on Condition, 330-32
  Trap on Overflow, 333-34

Exception(s), floating point, 364-73
  condition codes (IEEE aware/IEEE Non-Aware), 351, 358(table)
  continuing from an exception, 370-73
  Coprocessor Protocol, 365
  Format Error, 365
  FPCR exception enable byte, 350, *351*, 352
  FPSR accrued exception byte, *355*, 356
  FPSR exception status byte, 354, *355*, 362, 366, 372
  IDLE state frame and, 362-63
  Inexact Result (INEX2/INEX1), 366, 371
  Line F Emulator, 365
  multiple (priority and combinations), 370
  Operand Error (OPERR), 368-69, 372
  Overflow (OVFL), 369-70, 373
  pre-instruction and mid-instruction, 364
  Privilege Violation, 365
  Signaling NAN (SNAN), 370
  TRAPV (FTRAPcc), 365
  Underflow (UNFL), 367-68, 372
  Unordered Condition (BSUN), 365-66, 371
  Zero Divide (DZ), 366-67, 371

Exception(s), MMU-generated, 568-70

Exception Enable Byte (FPCR), 350, *351*, 362, 364

Exception stack frame, *54*

Exception Status Byte (FPSR), 354, *355*, 362, 366, 372. *See also* Branch/Set on Unordered (BSUN) enable bit (FPCR)

e^x function, 422-24

EXG instruction (Exchange Registers), 200-201

Exponent, 344-45
  2^x, 534-36
  10^x, 525-27
  get, with FGETEXP instruction, 428-30
  scale
    add source operand integer to destination, 491-93
    evaluation floating point condition, 494-96
EXTB instruction (Sign Extend-Byte->Long), 204-5
Extended precision format, 344, *345*, 346
  quantities converted to, 428, 431
Extension word of addressing modes, 22, 23-24, 25, 26-27, 29, 30, 31
External device, translate addresses provided by, 543
EXT instruction (Sign Extend Byte->Word and Word->Long), 202-3

FABS instruction (Floating Absolute Value), 386-88
FACOS instruction (Floating Arc Cosine function), 389-91
FADD instruction (Floating Add), 392-94
F<aop> arithmetic operation notation, 376
FASIN instruction (Floating Arc Sine function), 395-97
FATANH instruction (Floating Hyperbolic Arc Tangent function), 401-3
FATAN instruction (Floating Arc Tangent function), 398-400
Faults (exceptions), 45, 46-58
  summary, 46(table)
FBcc instruction (Branch on Floating Point Condition), 351, 356, 404-6
FCMP instruction (Floating Compare), 407-9
FCOSH instruction (Floating Hyperbolic Cosine function), 413-15
FCOS instruction (Floating Cosine function), 410-12
FDBcc instruction (Test Floating Point Condition, Decrement, and Branch), 351, 356, 416-18
FDIV instruction (Floating point divide), 419-21
FETOX instruction (ex Function), 422-24
FETOXM1 instruction (ex-1 Function), 425-27
FGETEXP instruction (Get Exponent), 428-30
FGETMAN instruction (Get Mantissa), 431-33
FINT instruction (Get Integer Part), 434-36
FINTRZ instruction (Get Integer Part, Round to Zero), 437-39
Floating point algorithm optimization, 661-73
  solving an equation by bisection, 661-65
  solving simultaneous linear equations, 665-73
Floating Point Control Register (FPCR), 350-53, 362, 434, 437, 455, 467
  enabling bits, 351-52
  Exception Enable Byte, *351*, 362, 364
  Mode Control Byte, 351, 352, *353*
Floating point coprocessor. *See* Coprocessor(s); Coprocessor conditions
Floating point instructions, description format, 379-85
  addressing modes, 382
  description section, 380
  example, 383-85
  headers, 379-80
  instruction format, 383
  numerical results, 381
Floating point Instruction Address Register (FPIAR), 356,371, 455, 467
Floating point modulo (FMOD), 354
Floating point number
  absolute value of, 386-88
  arc cosine of, 389-91
  arc sine of, 395-97
  arc tangent of, 398-400
  cosine of, 410-12
  hyperbolic arc tangent of, 401-3
  hyperbolic cosine of, 413-15
  hyperbolic sine of, 510-12
  hyperbolic tangent of, 522-24
  simultaneous sine and cosine, 506-9
  sine of, 503-5
  tangent of, 519-21
Floating Point Status Register (FPSR), 353-56, 362, 455, 467

  Condition Code byte, *353*
  Exception Status byte, 354, *355*, 362, 366, 372
  Quotient byte, *354*
Floating point unit (FPU) architecture, 343-79
  concurrency considerations, 373-78
  conditional operations, 356-60
  data formats, 344-50
  exceptions, 364-73
  register set, 350-56
  state frames, 360-64
  system software considerations, 378-79
Floating point unit (FPU) synchronization, 480
FLOG2 instruction (Logarithm, Base 2), 443-45
FLOG10 instruction (Logarithm, Base 10), 440-42
FLOGN instruction (Logarithm, Base e), 446-48
FLOGNP1 instruction (Logarithm (x+1), Base e), 449-51
FMOD instruction (Modulo Remainder), 452-54
FMOVECR instruction (Move Constant to Floating Point Data Register), 464-66
  minimal concurrency and, 375-76
FMOVE instruction, 355, 368
  full concurrency and, 377
FMOVE instruction-Control Reg (Move Floating Point Control Register), 455-57
FMOVE instruction-Data Reg (Move Floating Point Data Register), 458-63
FMOVEM instruction, minimal concurrency and, 375-76
FMOVEM instruction-Control Reg (Move Multiple Floating Point Control Registers), 467-69
FMOVEM instruction-Data Reg (Move Multiple Floating Point Data Registers), 470-73
FMUL instruction (Floating Point Multiple), 474-76
FNEG instruction (Floating Point Negate), 477-79
FNOP instruction (Floating Point No Operation), 480
Format code, exception identification with, 46, 298-99
  possible values, 299(table)
Format Error Exception, 57
  floating point, 365
Format word, 46, 298-99
Freeze cache bit, CACR, 38, 43
Freeze Data Cache bit, CACR, 43
Freeze Instruction Cache bit, CACR, 44
FREM instruction (IEEE Remainder), 481-84
FRESTORE instruction-Privileged (Restore Coprocessor State), 360, 362, 365, 485-87, 566
  continuing from exceptions with, 370-71
FSAVE instruction-Privileged (Save Coprocessor State), 360, 488-90, 566
  continuing from exceptions with, 370-71
FSCALE instruction (Scale Exponent), 491-93
FScc instruction (Scale Exponent), 351, 356, 494-96
FSGLDIV instruction (Floating Point Divide Single Precision), 367, 369, 497-99
FSGLMUL instruction (Floating Point Multiple Single Precision), 367, 369, 500-502
FSINCOS instruction (Simultaneous Sine and Cosine functions), 506-9
FSINH instruction (Floating Hyperbolic Sine function), 510-12
FSIN instruction (Floating Sine function), 503-5
FSQRT instruction (Floating Square Root), 513-15
FSUB instruction (Floating Subtract), 516-18
FTANH instruction (Floating Hyperbolic Tangent function), 522-24
FTAN instruction (Floating Tangent function), 519-21
FTENTOX instruction (10^X), 525-27
FTRAPcc instruction (Trap on Floating Point Condition), 351, 356, 528-30, 607

FTST instruction (Floating Text), 531-33

FTWOTOX instruction (2^x), 534-36

Function Code Lookup enable bit (TC register), 549-50

Function code pins FC0, FC1, and FC2, and memory request status, 17

Fundamentals of Interactive Computer Graphics, 652

gauss() function routines, 665-73

Gaussian elimination algorithm, 665

Global data access
  Apple MPW C language, 620-21
  Macintosh Pascal, 625
  UNIX pcc compiler, 621-23

Graphics routines optimization, 652-61
  Bresenham circle algorithm, 657-61
  Bresenham line algorithm, 653-57

Guard bit, 347

Heap, 540

Hexadecimal conventions, 4

Hidden bit, 345

Hyperbolic arc tangent function, 401-3

Hyperbolic cosine function, 413-15

Hyperbolic sine function, 510-12

Hyperbolic tangent function, 522-24

Idle Format State Frame, 566, *567*

IDLE state frame, 362-64, 371

IEEE aware exceptions, 358

IEEE non-aware exceptions, 358, 366

IEEE remainder instructions (FREM), 354, 481-84

IEEE Standard for Binary Floating-Point Arithmetic, 343

Illegal instruction (Take Illegal Instruction Exception), 206

Illegal instruction exception, 53-54, 206

Immediate data
  add to effective address operand, 76-77
  AND operation on effective address operand using, 84-85
  compare, with effective address operand, 159-61
  move, into a data register, 244-45
  OR operation on effective address operand using, 193, 272
  subtract, from an effective address operand, 316-17, 318-20

Immediate Data addressing mode, 36, 63

Immediate operand
  add to effective address operand, 78-79
  add to the stack pointer, 213-14
  AND operation on Condition Code Register and, 86-87
  AND operation on Status Register and, 88-89

InCircuit Emulator, 53

Indexing, 9

Index register scale factor in addressing modes, 22-23, 24, 25-26, 29, 33

Index Suppress (IS) flag, 25, 27, 31, 33

Indirect descriptor, 558, 562, *563*

Inexact Decimal Conversion (INEX1) enable bit (FPCR), 352

Inexact Result (INEX2) enable bit (FPCR), 352

Inexact Result (INEX2/INEX1) exception, 366
  continuing from, 371

Infinity formats, 346, 348-49
  binary representation of infinity, *348*

Initial Shift count (TC register), 551

Inside Macintosh, Volume II, 620

Instruction(s)
  description format, 65-69
    addressing modes section, 67-68
    description section, 66-67
    example section, 68-69
    header section, 65-66
    instruction format section, 68
    syntax symbols, 66
  description format, floating point, 379-85
  description format, MMU, 570-74
  specific instructions, 70-341 (*See also* names of individual instructions)
  specifying location of data to be used (*See* Addressing modes)
  tracer routine for, 676-83

Instruction Burst Enable flag, 44
Instruction cache
  68020 chip, 9
  68030 chip, *40*
  vs. data cache, 37
Instruction format, coprocessor, *62*-63
Instruction pipeline, 266
Integer bit, 345
Integer data types, 10-12, 344, 618
Integer part, convert source operand to extended precision and get the, 434-36
Internal Data format, 347-48
Interrupt(s), 45, 58-61
  non-maskable, 45, 59
  throwaway stack frame, *61*
  vectored vs. autovectored, 58-59
  vectors, 59(table)
Interrupt acknowledge cycle, 58
Interrupt handling, floating point, 379
Interrupt mask, 15, 58
Interrupt priority, 45
Interrupt Stack Frames, 60-*61*
Interrupt Stack Pointer (ISP), 16, 61
Invalid descriptor, 558, *562*
IPL2, IPL1, and IPL0 pins, 58
is020() routines, 673-74

JMP instruction (Jump to Effective Address), 207-8
JSR instruction (Jump to Subroutine at Effective Address), 209-10, 295, 616
  reverse with RTS instruction, 304

LEA instruction (Load Effective Address), 211-12
Line A Emulator Trap, 55-56
Line algorithm, 653-57
Line F Emulator exception, 56
  floating point, 365
  MMU-generated, 569
LINK instruction (Link and Allocate), 213-14
  reverse action of, 337-38
Little endian byte order, 11
Logarithm
  calculate with base 2, 443-45
  calculate with base 10, 440-42
  calculate with base e, 446-48
  calculate with base e, (x+1), 449-51
Logical address, 538, 539, *542*
  test, 603-6
  validate, 610-12
Long word
  defined, 11
  floating point, 344
Loop
  conditional MMU, 565-66, 578-80
  preventing infinite BSUN, 371
  termination condition, 177
  terminating floating point condition, 416-18
  unrolling for performance optimization, 377-78, 632-40
    single-byte versions, 632-34
    unrolled versions, 634-40
Loop mode feature, 68010 chip, 8
LSL instruction (Logical Shift Left), 215-17
LSR instruction (Logical Shift Right), 218-20

Macintosh Pascal, assembly interface with, 623-25
  data types, 623-24
  global data access, 625
  procedure calling standard, 624-25
Macintosh Programmer's Workshop (MPW), 3
  C language interface with assembly
    data types, 614-16
    global data access, 620-21
    procedure calling standard, 616-20
  optimization examples using 627-84
  use of FTEST instruction, 531
  use of FTPcc instruction, 528
Macintosh programming model, 2-5
Main processor
  synchronization, 480
  test condition on, decrement, and branch, 177-79
malloc function (C language), 540
Mantissa, 344
  get, 431-33

Master Stack Pointer (MSP), 16, 61, 613
Mathematical operations. *See* Addition; Division; Multiplication; Subtraction
Matrix triangularization code, 667, 670
Memory
  add space in, with NOP instruction, 266-67
  locations in
    add two, 80-81
    compare two, 162-63
    subtract two, 321-23
  move multiple registers to and from, 237-40
Memory addressing modes. *See* Addressing modes
Memory cache operation. *See* Cache memory
Memory clear routine, 636-37
Memory Indirect Post-Indexed addressing mode, 25-26
Memory Indirect Pre-Indexed addressing mode, 27, *28*
Memory Management Unit (MMU), 537-612
  68010 support for, 8
  architecture (*see* Memory Management Unit architecture)
  background, 537
  defined, 537, *538*
  instruction description format, 570-74
    addressing modes, 571-72
    header, 571
    instruction format, 572-74
  instructions, 575-612
  move data across spaces managed by, 246-48
  restore state of, 595-96
  save state of, 597-99
  techniques of, 538-42
    address translation cache, 541-42
    base/bounds, 538, 539-41
Memory Management Unit Access Violation exception, 570
Memory Management Unit architecture, 542-70
  68030 chip, 544
  68451 chip, 543
  68851 chip, 543
  access control registers (CAL, VAL, SCC, AC), 555-57
  breakpoint registers, 554-55
  cache status register, 557
  conditional operations, 565-66
  exceptions generated by MMU, 568-70
  register set, 546-52
  root pointer table, 552-53
  state frames, 566-68
  translation table
    descriptors, 558-65
    overview, 544, *545*
  transparent translation registers, 553-54
Memory Management Unit Configuration exception, 570
Memory Management Unit Illegal Operation exception, 570
Memory Managment Unit Status Register (MMUSR), *547*-48
  check state of a single, 601
  set condition codes in, 603-6
Memory protection with CALLM instruction, 135-40, 555-57
Mid-Coprocessor Format State Frame, *567*
Mid-instruction exception, 364
MIT syntax, 3
Mode Control Byte (FPCR), 351, 352, *353*
Module
  call with CALLM instruction, 9, 10, 135-40
    Module Call Stack Frame format, *137*
    module descriptor types, 136, 138-40
  return from, 9, 10, 300-301
Modulus calculation, 452-54, 481-84
Monadic arithmetic operations, 374, 376, 381
Motorola assembly language standard, 3
  hexadecimal conventions, 4
MOVEA instruction (Move Data from Source to Address Register), 221, 223-24
MOVEC instruction-Privileged (Move Control Register), 38, 42, 235-36
MOVE From CCR instruction (Move Data from Condition Code Register to Effective Address), 225-26
MOVE From SR instruction-Privileged (Move Data from Status Register to Effective Address), 229-30

MOVE instruction (Move Data from Source to Destination), 221-22

MOVEM instruction (Move Multiple Registers), 237-40

MOVEP instruction (Move Peripheral Data), 241-43
  hardware configuration, *242*

MOVEQ instruction (Move Quick), 221, 244-45

MOVES instruction-Privileged (Move Address Space), 17, 246-48

MOVE To CCR instruction (Move Data to Condition Code Register from Effective Address), 227-28

MOVE To SR instruction-Privileged (Move Data to Status Register from Effective Address), 231-32

MOVE USP instruction-Privileged (MOVE User Stack Pointer), 233-34

MULS instruction (Signed Multiply-Word Form), 249-50

MULS.L instructions (Signed Multiply-Long Forms), 251-53

Multiple-precision operations, 162, 259, 264, 305, 321

Multiplication
  signed (long forms), 251-53
  signed (word form), 249-50
  source and destination operands, 474-76, 500-502
  unsigned (long forms), 256-58
  unsigned (word form), 254-55

Multitasking
  context switching and (*See* Context switching)
  instructions useful in, 233, 246
  protected environment for, 537
  role of CACR Clear Instruction Cache bit, 44
  virtual memory and, 537

MULU instruction (Unsigned Multiply-Word Form), 254-55

MULU.L instructions (Unsigned Multiply-Long Forms), 256-58

Natural logarithm, 446, 449

NBCD instruction (Negate Decimal with Extend), 259-61

NEG instruction (Negate Binary), 262-63

NEGX instruction (Negate Binary with Extend), 264-65

new function (Pascal), 540

Non-maskable interrupt, 45, 59

NOP instruction (No Operation), 266-67
  floating point, 480

Normalized binary floating-point numbers, 345

Not-A-Number (NAN) format, 346, 349-50
  BCD representation of, *350*
  binary representation of, *350*
  condition codes and, 356, 358

Notational conventions, 2-5
  for floating point condition codes, 356

NOT instruction (One's Complement), 268-69

NOT logical operator, floating point, 356

NULL state frame, 360, 362

Numerical Methods with FORTRAN, 665

One's complement, 268-69

Operand(s). *See also* Destination operand; Effective address operand; Immediate operand; Source operand, floating point
  add, to a floating point data register, 392-94
  compare and swap one, 141-43
  compare and swap two, 144-45
  test, 335-36
  test floating point, 531-33

Operand error enable bit (FPCR), 352, 367, 368

Operand Error (OPERR) exception, 368-69
  continuing from, 372

ORI instruction (Inclusive OR Immediate), 272-73

OR instruction (Inclusive OR Logical), 270-71

ORI to CCR instruction (Inclusive OR Immediate to Condition Code Register), 274-75

ORI to SR instruction-Privileged (Inclusive OR Immediate to Status Register), 276-77
OR logical operator
  on Condition Code Register
    using EORI to CCR instruction, 196-97
    using ORI to CCR instruction, 274-75
  on data register and effective address operand
    using EOR instruction, 191-92
    using OR instruction, 270-71
  on effective address operand using immediate data
    using EORI instruction, 193-95
    using ORI instruction, 272-73
  floating point sticky bit as, 348
  on Status Register, 198-99
OR logical operator, floating point, 356
Outer displacement, 25, *26*, 32
Overflow, defined, 348
Overflow (OVFL) enable bit (FPCR), 352
Overflow (OVFL) exception, 369-70
  catastrophic, 373
  continuing from, 373

Packed Decimal String Real Data format, *346*-47
PACK instruction (Pack BCD), 278-79
Page, defined, 541
Page descriptors, 558, *560*-62
Paged Memory Management Unit (PMMU), 10
  BUSERR exception (page fault), 47
Paged Memory Mangement Unit Status Register (PSR), *547*-48
Page fault, 47
Page Size bit (TC register), 550
Page table, defined, 542
Pascal, assembly interface with, 623-25
PBcc instruction-Privileged (Branch on MMU Condition), 565, 575-77
PDBcc instruction-Privileged (Test MMU Condition, Decrement, and Branch), 565, 578-80
PEA instruction (Push Effective Address), 280-81
Performance optimization, 626-84
  68882 chip, 377-78
  guidelines, 626-27
  programming examples, 627-84
PFLUSH instruction-Privileged (Flush MMU ATC Entries), 581-84
PFLUSHR instruction-Privileged (Flush ATC and RPT Entries), 585-87
Physical address, 538, 539, *542*
  segment, *540*
  validate, 610-12
pixel() routine, 654-55
PLOAD instruction-Privileged (Load ATC Entry), 569, 588-90
PMOVE instruction-Privileged (Move MMU Register), 555, 591-94
Pop operation, 20
Postincrement addressing mode, 20, 240
Predecrement addressing mode, 21, 240
Pre-instruction exception, 364
PRESTORE instruction-Privileged (Restore MMU State), 555, 566, 595-96
printf function (C language), 616
Privileged instructions, 88, 170, 172, 198, 229, 231, 233, 235, 246, 276, 282, 297, 310
  floating point, 485, 488
Privilege Violation exception, 55
  floating point, 365
Procedure calling standard
  Apple MPW C, 616-20
  Macintosh Pascal, 624-25
  UNIX pcc compiler, 622
Processor. *See also* Coprocessor(s); Main processor modes, 15 (*See also* Supervisor (privileged) software; User mode)
  synchronizing multiple, 325-27
Program counter (PC), 12
  exception processing and, 45-46, 297-99
  instructions which alter the value of, 96, 167, 168, 177
  memory addressing relative to, 3
  pop longword on top of the stack into, 295-96
  zero suppressed, 31, 33

Program Counter Indirect with Displacement addressing mode, 27-28, *29*

Program Counter Indirect with Index (8-bit displacement) addressing mode, 28-*30*

Program Counter Indirect with Index (Base Displacement) addressing mode, 30-32

Program Counter Memory Indirect Post-Indexed addressing mode, 32-34

Program Counter Memory Indirect Pre-Indexed addressing mode, *34*-35

Program counter register, 13

Programming error. *See* Exception(s)

Programming in assembly language. *See* Assembly language programming

PSAVE instruction-Privileged (Save MMU State), 597-99
  MMU state frames and, 566, 567

PScc instruction-Privileged (MMU Set Conditionally), 600-602

PTEST instruction-Privileged (Test Logical Address), 569, 603-6

PTRAPcc instruction-Privileged (Trap on MMU Condition), 607-9

Push operation, 21

PVALID instruction-Privileged (Validate Address), 610-12

qsort optimization, 640-48

Quotient byte (FPSR), *354*

Read/modify/write bus cycle, 141-43

Read/modify/write memory access operation, 325-27

Register(s). *See also* Address registers; Control registers; Control registers, floating point; Data register(s); Destination Function Code (DFC) register; Source Function Code (SFC) register; Status register (SR)
  base/bounds memory management, 539
  exchange contents of, 200-201
  memory addressing relative to, 3
  move multiple, to and from memory, 237-40
  usage, 4

Register A5, 4, 620

Register A6, 618

Register A7, 13, 14, 16, 337
  push and pop operations with, 20, 21

Register set, 12-17
  68020 chip, *12*
  address registers, 13
  cache address register (CAAR), 17, 39, 42
  cache control register (CACR), 17, 38-39, 42-44
  condition code register, 14
  data registers (*See* Data register(s))
  destination function code (DFC), 17
  FC0, FC1, and FC2 pins and memory request states, 17
  interrupt stack pointer (ISP), 16, 61
  master stack pointer (MSP), 16, 61
  processor modes, 15
  program counter register, 13
  source function code register (SFC), 17
  status register, 14-15
  vector base register (VBR), 16

Register set, floating point, 350-56
  68881/68882 chips, 350, *351*
  Floating Point Control Register (FPCR), 350-53
  Floating Point Instruction Address Register (FPIAR), 356
  Floating Point Status Register (FPSR), 353-56

Register set, Memory Management Unit, 546-52
  diagram of, *546*
  move MMU registers, 591-94
  root pointer table, 552-53
  status register, 547-48
  translation control, 548-51
  translation table root pointers, 551-52

Reset Format State Frame, *566*

RESET instruction-Privileged (Reset External Devices), 282

RESET interrupt, 59-60, 282

ROL instruction (Rotate Left without Extend), 283-85

Root pointer, 544, *545*, 550
  bit description, 551-52
  format, *551*

Root Pointer Table (RPT), 552-53, 557
  flush entries from ATC and, 585-87

ROR instruction (Rotate Right without extend), 286-88

Rotate operations
  left with extend, 289-91
  left without extend, 283-85
  right with extend, 292-94
  right without extend, 286-88

Round bit, 347

Rounding modes supported by 68881/68882 coprocessors, 347-48
  effect on FINT instruction, 434-36
  effect on Overflow (OVFL) exception, 369-70
  Round to Minus Infinity, 347-48
  Round to Nearest, 347
  Round to Positive Infinity, 348
  Round to Zero (chop mode), 347, 437, 491

ROXL instruction (Rotate Left with Extend), 289-91

ROXR instruction (Rotate Right with Extend), 292-94

RTD instruction (Return and Deallocate), 295-96

RTE instruction-Privileged (Return from Exception), 46, 53, 297-99, 302, 370-71

RTM instruction (Return from Module), 135, 300-301
  access control registers supporting, 555-57

RTR instruction (Return and Restore Condition Codes), 302-3

RTS instruction (Return from Subroutine), 131, 304

SBCD instruction (Subtract Binary Coded Decimal with Extend), 305-6

Scaling, 9
  is020() routine for optimizing, 673-74

scanf function (C language), 616

Scatter/gather I/O, 543

Scc instruction (Set According to Condition), 307-9
  conditions, 308-9

Searching routine optimization, 648-52
  assembly language optimization, 650-52
  C implementation, 649-50
  performance results, 642

Segment in physical memory, *540*, 541
  fixed length (*See* Page)

Signaling NAN (SNAN) exception, 370
  continuing from, 373

Signaling NAN (SNAN) format (FPCR), 349, 352

Sign extension (changing upper bits of the register), 13, 21-22
  perform on data register, 202-3

Simultaneous linear equation solutions, optimization, 665-73
  assembly language implementation, 667-73
  C language implementation, 665-67

Sine function, 503-5
  simultaneous with cosine, 506-9

Single precision format, 344, *345*, 346, 497, 500

SNAN. *See* Signaling NAN (SNAN) exception

Software Tools in Pascal, 640

Sort routine optimization, 640-48
  assembly language implementation, 642-47
  C implementation, 641-42
  performance results, 648

Source Function Code (SFC) register, 17, 246

Source operand, floating point, 373-74
  calculate 2^x, 534-36
  calculate 10^x, 525-27
  calculate logarithm
    with base 2, 443-45
    with base 10, 440-42
    with base e, 446-48
    with base e, x+1, 449-51
  convert to integer and add to destination exponent, 491-93
  divide data register by, 419-21, 497-99
  divide destination operand by, 452-54
  extract binary exponent with, 428-30
  extract integer part, 434-36
  extract integer part and round to zero, 437-39

  - get mantissa, 431-33
  - multiply with destination operand, 474-76, 500-502
  - negative of, placed in data register, 477-79
  - set condition codes by subtracting, from data register, 407-9
  - set condition codes by testing, 531-33
  - square root of, placed in data register, 513-15
  - subtract, from data register, 516-18
- Special Status Register
  - 68010 chip, *49*
  - 68020/30 chip, 50-51, *52*
- Spurious interrupt exception, 58
- Spurious interrupt vector, 60
- Square root, floating point, 513-15
- Stack Change Control (SCC) register, 555, *556*
- Stack frame(s), 213
  - exception, 45-46
    - for BUSERR and Addressing Error, 48-53
    - format 0 (default), *45*, 206, 328
    - format 1 (throwaway), 60, *61*
    - format 2 (division by zero), *54*, 607
    - format 9 (coprocessor protocol), 56, *57*, 370
    - long and short, 50, *51*
    - Module Call, *137*
  - procedure calling standard and, 618, *619*
  - RTE instruction format and, 46, 298-99
- Stack pointer, 15
- Stack pointer register (A7), 13, 16. *See also* Interrupt Stack Pointer (ISP); Master Stack Pointer (MSP)
- State frame(s), floating point, 360-64
  - 68881 and 68882 chip, *361*
  - busy state frame, 360, 364
  - idle state frame, 360, 362-64
  - null state frame, 360-62
- State frame(s), Memory Management Unit, 566-68
- Status Register (SR), 14-*15*. *See also* Special Status Register
  - AND operation on immediate operand and, 88-89
  - copy, into effective address, 229-30
  - copy data from effective address operand into, 227-28
  - exception processing and, 45-46, 61, 297
  - floating point, 353-56
  - instruction tracer bit in, 676-83
  - load, and suspend execution, 310-11
  - move data to, from effective address operand, 231-32
  - move low order 8 bits of, into effective address, 225-26
  - OR operations on, 198-99, 276-77
- Status Register, Memory Management Unit. *See* Memory Management Unit Status Register (MMUSR); Paged Memory Management Unit Status Register (PSR)
- Sticky bit, 347
- STOP instruction-Privileged (Load Status Register and Stop), 310-11
- strcmp() C string function performance optimization, 630-32
- strcpy() C string function performance optimization, 627-30
- String(s)
  - C, 616, 624
  - C string function performance optimization examples, 627-32
  - Pascal, 624
- SUBA instruction (Subtract Address), 314-15
- SUBI instruction (Subtract Immediate), 312, 316-17
- SUB instruction (Subtract Binary), 312-13, 322
- SUBQ instruction (Subtract Quick), 318-20
- Subroutine
  - branch to, 131-32
  - jump to, at effective address, 209-10
  - return from, 304
  - return from, and deallocate, 295-96
- Subtraction
  - 2's complement binary numbers, 312-13
  - BCD numbers with eXtend condition, 305-6
  - binary numbers with eXtend condition, 321-23
  - destination operand and extend bit from zero, 264-65

  destination operand and X condition code bit from zero, 259-61
  destination operand from zero, 262-63
  effective address operand from address register, 314-15
  immediate data from effective address operand, 316-17, 318-20
  source operand from data register, 516-18
SUBX instruction (Subtract Binary with eXtend), 321-23
Sun workstations, 2, 3
Supervisor Root Enable bit (TC register), 549, 550
Supervisor Root Pointer (SRP), 550, 551-52
Supervisor (privileged) software, 15, 16
  cache control operations under, 38
  registers, 68020 chip, *12*
Supervisor Stack Pointer (SSP), 16, 60
SWAP instruction (Swap Words of a Data Register), 324
Symbolic debugger. *See* Debugging software

Table descriptors, *558*-60
Tags, cache memory, 37
Tangent function, 519-21
TAS instruction (Test and Set), 325-27
Throwaway stack frame, 60, *61*
Tracer routine to trace instructions, 676-83
  68000 implementation, 680-83
  68010/020/030 implementation, 677-80
  extensions, 683
Trace Trap, 55, 310
Transcendental functions, 348
Translation Control (TC) register, 548-51
  bits description, 548-51
  layout, *549*
Translation lookaside buffer (TLB), 541
Translation table, Memory Management Unit
  descriptors, 558-65
    early termination, 563-65
    indirect, 562-63
    invalid, 562
    page, 560-62
    table, 558-60
  overview, 544, *545*
Transparent translation registers, 553-54
TRAPcc instruction (TRAP on condition), 176, 330-32, 607
  conditions, 331
TRAP exception, 57-58, 328-29
  on condition, 330-32
  on floating point condition, 528-30
  on MMU condition, 607-9
  on overflow, 333-34
TRAP instruction (Take TRAP exception), 328-29
TRAPV (FTRAPcc) exception, 55, 365
TRAPV instruction (TRAP on Overflow), 55, 333-34
Triangular form, matrix, 667, 670
TST instruction (Test Operand), 335-36

Underflow (UNFL) enable bit (FPCR), 352, 367, 372
Underflow (UNFL) exception, 367-68
  catastrophic, 372
  continuing from, 372
Uninitialized interrupt vector, 60
UNIX assembly language standard, 3
UNIX Portable C Compiler, assembly interface with, 621-23
  data types, 622
  global data access, 622-23
  procedure calling standard, 622
UNLK instruction (Unlink), 337-38
Unnormalized binary floating point numbers, 346
Unordered Condition (BSUN) exception, 365-66. *See also* Branch/Set on Unordered (BSUN) enable bit (FPCR)
  preventing infinite loops, 371
UNPK instruction (Unpack BCD), 339-41
User mode, 15, 16
  registers, 68020 chip, *12*
User Stack Pointer (USP), 16
  move and set, 233-34

Validate Access Level (VAL) register, *555*, 610

Variables, allocate stack space for local, with LINK, 213-14

Vector(s), 16, 44-45
  interrupt, 58, 59(table)

Vector Base Register, 16, 44, 46

Vector table, 16

Virtual memory, 2, 8
  multitasking and, 537
  page faults and, 47

Word
  defined, 11
  floating point, 344

Write Allocate bit, CACR, 43

Write allocation, 41-42

Write-through operation, 41

Zero
  division by, 54, 366-67
  set effective address operand to, 151-52
  subtract destination operand and X condition code bit from, 259-61
  subtract destination operand from, 262-63

Zero Divide exception. *See* Division by Zero (DZ) exception

Zero suppressed address register (ZAn) , 25, 27

Zero suppressed PC (ZPC), 31, 33

> $29.95 FPT USA

68030 Assembly Language Reference

Includes the 68020

Steve Williams

The Complete Reference Guide to the Entire 68000 Family of Microprocessors

This is the essential programmer's reference to Motorola's hot 32-bit microprocessors, the 68030 and the 68020, as well as the rest of the 68000 family. The powerful 68030 is the heart of such leading-edge computers as the Macintosh® SE 30, the Macintosh IIx, and the NeXT™ Computer System, while the 68020 powers the Macintosh II. Programmers developing software for any of these computers will find *68030 Assembly Language Reference* to be an indispensable guide.

68030 Assembly Language Reference provides detailed and comprehensive information on programming the 68030 and 68020 chips, including thorough discussions of:

- the 68030 and 68020 architecture
- floating-point arithmetic
- the Paged Memory Management Unit (PMMU)
- the separate data and instruction caches

In addition, more than half of the book is devoted to actual instruction sets for all assembly language operations. Numerous programming examples give readers hands-on experience in learning to program with the 68030 and 68020 microprocessors.

Steve Williams is an established authority on the entire 68000 family of chips. He was the architect and project manager for a 32-bit operating system at Digital Research, Inc., and is currently employed by Apple® Computer, Inc., as a software architect. He is also the author of two best-selling assembly language books, *Programming the 68000* and *Programming the Macintosh in Assembly Language.*

Cover design by Doliber Skeffington

Addison-Wesley
Publishing Company, Inc.

52995

9 780201 088762

ISBN 0-201-08876-2